The 30 Year Horizon

Manuel Bronstein
James Davenport
Albrecht Fortenbacher
Jocelyn Guidry
Michael Monagan
Jonathan Steinbach
Stephen Watt

William Burge
Michael Dewar
Patrizia Gianni
Richard Jenks
Scott Morrison
Robert Sutor
Jim Wen

Timothy Daly
Martin Dunstan
Johannes Grabmeier
Larry Lambe
William Sit
Barry Trager
Clifton Williamson

Volume 10: Axiom Algebra: Packages
Portions Copyright (c) 2005 Timothy Daly

The Blue Bayou image Copyright (c) 2004 Jocelyn Guidry

Portions Copyright (c) 2004 Martin Dunstan
Portions Copyright (c) 2007 Alfredo Portes
Portions Copyright (c) 2007 Arthur Ralfs
Portions Copyright (c) 2005 Timothy Daly

Portions Copyright (c) 1991-2002,
The Numerical ALgorithms Group Ltd.
All rights reserved.

This book and the Axiom software is licensed as follows:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of The Numerical ALgorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

<table>
<thead>
<tr>
<th>Michael Albaugh</th>
<th>Cyril Alberga</th>
<th>Roy Adler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christian Aistleitner</td>
<td>Richard Anderson</td>
<td>George Andrews</td>
</tr>
<tr>
<td>S.J. Atkins</td>
<td>Henry Baker</td>
<td>Martin Baker</td>
</tr>
<tr>
<td>Stephen Balzac</td>
<td>Yurij Baransky</td>
<td>David R. Barton</td>
</tr>
<tr>
<td>Gerald Baumgartner</td>
<td>Gilbert Baumslag</td>
<td>Michael Becker</td>
</tr>
<tr>
<td>Nelson H. F. Beebe</td>
<td>Jay Belanger</td>
<td>David Bindel</td>
</tr>
<tr>
<td>Fred Blair</td>
<td>Vladimir Bondarenko</td>
<td>Mark Botch</td>
</tr>
<tr>
<td>Raoul Bourquin</td>
<td>Alexandre Bouyer</td>
<td>Karen Braman</td>
</tr>
<tr>
<td>Peter A. Broadbery</td>
<td>Martin Brock</td>
<td>Manuel Bronstein</td>
</tr>
<tr>
<td>Stephen Buchwald</td>
<td>Florian Bundschuh</td>
<td>Luanne Burns</td>
</tr>
<tr>
<td>William Burge</td>
<td>Ralph Byers</td>
<td>Quentin Carpent</td>
</tr>
<tr>
<td>Robert Cavniness</td>
<td>Bruce Char</td>
<td>Ondrej Certik</td>
</tr>
<tr>
<td>Tzu-Yi Chen</td>
<td>Cheekai Chin</td>
<td>David V. Chudnovsky</td>
</tr>
<tr>
<td>Gregory V. Chudnovsky</td>
<td>Mark Clements</td>
<td>James Cloos</td>
</tr>
<tr>
<td>Jia Zhao Cong</td>
<td>Josh Cohen</td>
<td>Christophe Couil</td>
</tr>
<tr>
<td>Don Coppersmith</td>
<td>George Corliss</td>
<td>Robert Corless</td>
</tr>
<tr>
<td>Gary Cornell</td>
<td>Meino Cramer</td>
<td>Jeremy Du Croz</td>
</tr>
<tr>
<td>David Cyganski</td>
<td>Nathaniel Daly</td>
<td>Timothy Daly Sr.</td>
</tr>
<tr>
<td>Timothy Daly Jr.</td>
<td>James H. Davenport</td>
<td>David Day</td>
</tr>
<tr>
<td>James Demmel</td>
<td>Didier Deshommes</td>
<td>Michael Dewar</td>
</tr>
<tr>
<td>Jack Dongarra</td>
<td>Jean Della Dora</td>
<td>Gabriel Dos Reis</td>
</tr>
<tr>
<td>Claire DiCrescendo</td>
<td>Sam Dooley</td>
<td>Lionel Ducos</td>
</tr>
<tr>
<td>Iain Duff</td>
<td>Lee Duhem</td>
<td>Martin Dunstan</td>
</tr>
<tr>
<td>Brian Dupee</td>
<td>Dominique Duval</td>
<td>Robert Edwards</td>
</tr>
<tr>
<td>Heow Eide-Goodman</td>
<td>Lars Erickson</td>
<td>Richard Fateman</td>
</tr>
<tr>
<td>Bertfried Fauser</td>
<td>Stuart Feldman</td>
<td>John Fletcher</td>
</tr>
<tr>
<td>Brian Ford</td>
<td>Albrecht Fortenbacher</td>
<td>George Frances</td>
</tr>
<tr>
<td>Constantine Frangos</td>
<td>Timothy Freeman</td>
<td>Korriin Fu</td>
</tr>
<tr>
<td>Marc Gaetano</td>
<td>Rudiger Gebauer</td>
<td>Van de Geijn</td>
</tr>
<tr>
<td>Kathy Gerber</td>
<td>Patricia Gianni</td>
<td>Samantha Goldrich</td>
</tr>
<tr>
<td>Holger Gollan</td>
<td>Teresa Gomez-Diaz</td>
<td>Laureano Gonzalez-Vega</td>
</tr>
<tr>
<td>Stephen Gortler</td>
<td>Johannes Grabmeier</td>
<td>Matt Grayson</td>
</tr>
<tr>
<td>Klaus Ebbe Grue</td>
<td>James Griesmer</td>
<td>Vladimir Grinberg</td>
</tr>
<tr>
<td>Oswald Gschnitzer</td>
<td>Ming Gu</td>
<td>Jocelyn Guidry</td>
</tr>
<tr>
<td>Gaetan Hache</td>
<td>Steve Hague</td>
<td>Satoshi Hamaguchi</td>
</tr>
<tr>
<td>Sven Hammarling</td>
<td>Mike Hansen</td>
<td>Richard Hanson</td>
</tr>
<tr>
<td>Richard Harke</td>
<td>Bill Hart</td>
<td>Vilya Harvey</td>
</tr>
<tr>
<td>Martin Hassner</td>
<td>Arthur S. Hathaway</td>
<td>Dan Hatton</td>
</tr>
<tr>
<td>Waldek Hebisch</td>
<td>Karl Hegbloom</td>
<td>Ralf Hemmecke</td>
</tr>
</tbody>
</table>
Henderson Antoine Hersen Roger House
Gernot Hueber Pietro Iglio Alejandro Jakubi
Richard Jenks William Kahan Kai Kaminski
Grant Keady Wilfrid Kendall Tony Kennedy
Ted Kosan Paul Kosinski Klaus Kusche
Bernhard Kutzler Tim Lahey Larry Lambe
Kaj Laurson George L. Legendre Franz Lehner
Frederic Lehobey Michel Levaud Howard Levy
Ren-Cang Li Rudiger Loos Michael Lucks
Richard Luczak Cammi Maguire Francois Maltey
Alasdair McAndrew Bob McElrath Michael McGetrick
Edi Meier Ian Meikle David Mentre
Victor S. Miller Gerard Milmeister Mohammed Mobarak
H. Michael Moeller Michael Monagan Marc Moreno-Maza
Scott Morrison Joel Moses Mark Murray
William Naylor Patrice Naudin C. Andrew Neff
John Nelder Godfrey Nolan Arthur Norman
Jinzhong Niu Michael O’Connor Summat Oemrawsingh
Kostas Diokonomou Humberto Ortiz-Zuazaga Julian A. Padget
Bill Page David Parnas Susan Pelzel
Michel Petitot Didier Pinchon Ayal Pinkus
Frederick H. Pitts Jose Alfredo Portes Gregorio Quintana-Orti
Claude Quitte Arthur C. Ralfs Norman Ramsey
Anatoly Raportirenko Albert D. Rich Michael Richardson
Guilherme Reis Huan Ren Renaud Riobooo
Jean Rivlin Nicolas Robidoux Simon Robinson
Raymond Rogers Michael Rothstein Martin Rubey
Philip Santas Alfred Scherhorn William Schelter
Gerhard Schneider Martin Schoener Marshall Schor
Frithjof Schulze Fritz Schwarz Steven Segletes
V. Sina Nick Simicich William Sit
Elena Smirnova Jonathan Steinbach Fabio Stumbo
Christine Sundaresan Robert Sutor Moss E. Sweedler
Eugene Surowitz Max Tegmark T. Doug Telford
James Thatcher Balbir Thomas Mike Thomas
Dylan Thurston Steve Toleque Barry Trager
Temos T. Tsikas Gregory Vanuxem Bernhard Wall
Stephen Watt Jaap Weel Juergen Weiss
M. Weller Mark Wegman James Wen
Thorsten Werther Michael Wester R. Clint Whaley
John M. Wiley Berhard Will Clifton J. Williamson
Stephen Wilson Shmuel Winograd Robert Wisbauer
Sandra Wityak Waldemar Wiwianka Knut Wolf
Liu Xiaojun Clifford Yapp David Yun
Vadim Zhnytkov Richard Zippel Evelyn Zoernack
Bruno Zuercher Dan Zwilling
Contents

1 Chapter Overview 1

2 Chapter A 3
   package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis 3
   AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO) 4
   package AFALGRES AffineAlgebraicSetComputeWithResultant 8
   AffineAlgebraicSetComputeWithResultant (AFALGRES) 9
   package AF AlgebraicFunction 13
   AlgebraicFunction (AF) 14
   package INTHERAL AlgebraicHermiteIntegration 19
   AlgebraicHermiteIntegration (INTHERAL) 19
   package INTALG AlgebraicIntegrate 22
   AlgebraicIntegrate (INTALG) 23
   package INTAF AlgebraicIntegration 30
   AlgebraicIntegration (INTAF) 31
   package ALGMANIP AlgebraicManipulations 33
   AlgebraicManipulations (ALGMANIP) 34
   package ALGMFACT AlgebraicMultFact 38
   AlgebraicMultFact (ALGMFACT) 39
   package ALGPKG AlgebraPackage 41
   AlgebraPackage (ALGPKG) 42
   package ALGFACT AlgFactor 51
   AlgFactor (ALGFACT) 52
   package INTPACK AnnaNumericalIntegrationPackage 55
   AnnaNumericalIntegrationPackage (INTPACK) 56
   package OPTPACK AnnaNumericalOptimizationPackage 66
   AnnaNumericalOptimizationPackage (OPTPACK) 67
   package ODEPACK AnnaOrdinaryDifferentialEquationPackage 76
   AnnaOrdinaryDifferentialEquationPackage (ODEPACK) 77
   package PDEPACK AnnaPartialDifferentialEquationPackage 85
   AnnaPartialDifferentialEquationPackage (PDEPACK) 86
   package ANY1 AnyFunctions1 92
   AnyFunctions1 (ANY1) 93
   package API ApplicationProgramInterface 94
### CONTENTS

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationProgramInterface (API)</td>
<td>101</td>
</tr>
<tr>
<td>package APPRULE ApplyRules</td>
<td>103</td>
</tr>
<tr>
<td>ApplyRules (APPRULE)</td>
<td>104</td>
</tr>
<tr>
<td>package APPLYORE ApplyUnivariateSkewPolynomial</td>
<td>107</td>
</tr>
<tr>
<td>ApplyUnivariateSkewPolynomial (APPLYORE)</td>
<td>108</td>
</tr>
<tr>
<td>package ASSOCEQ AssociatedEquations</td>
<td>109</td>
</tr>
<tr>
<td>AssociatedEquations (ASSOCEQ)</td>
<td>110</td>
</tr>
<tr>
<td>package PMPRED AttachPredicates</td>
<td>112</td>
</tr>
<tr>
<td>AttachPredicates (PMPRED)</td>
<td>113</td>
</tr>
<tr>
<td>package AXSERV AxiomServer</td>
<td>114</td>
</tr>
<tr>
<td>AxiomServer (AXSERV)</td>
<td>115</td>
</tr>
<tr>
<td>3 Chapter B</td>
<td>133</td>
</tr>
<tr>
<td>package BALFACT BalancedFactorisation</td>
<td>133</td>
</tr>
<tr>
<td>BalancedFactorisation (BALFACT)</td>
<td>134</td>
</tr>
<tr>
<td>package BOP1 BasicOperatorFunctions1</td>
<td>135</td>
</tr>
<tr>
<td>BasicOperatorFunctions1 (BOP1)</td>
<td>136</td>
</tr>
<tr>
<td>package BEZIER Bezier</td>
<td>139</td>
</tr>
<tr>
<td>Bezier (BEZIER)</td>
<td>157</td>
</tr>
<tr>
<td>package BEZOUT BezoutMatrix</td>
<td>158</td>
</tr>
<tr>
<td>BezoutMatrix (BEZOUT)</td>
<td>159</td>
</tr>
<tr>
<td>package BLUPACK BlowUpPackage</td>
<td>163</td>
</tr>
<tr>
<td>BlowUpPackage (BLUPACK)</td>
<td>164</td>
</tr>
<tr>
<td>package BOUNDZRO BoundIntegerRoots</td>
<td>169</td>
</tr>
<tr>
<td>BoundIntegerRoots (BOUNDZRO)</td>
<td>170</td>
</tr>
<tr>
<td>package BRILL BrillhartTests</td>
<td>172</td>
</tr>
<tr>
<td>BrillhartTests (BRILL)</td>
<td>174</td>
</tr>
<tr>
<td>4 Chapter C</td>
<td>177</td>
</tr>
<tr>
<td>package CARTEN2 CartesianTensorFunctions2</td>
<td>177</td>
</tr>
<tr>
<td>CartesianTensorFunctions2 (CARTEN2)</td>
<td>178</td>
</tr>
<tr>
<td>package CHVAR ChangeOfVariable</td>
<td>179</td>
</tr>
<tr>
<td>ChangeOfVariable (CHVAR)</td>
<td>180</td>
</tr>
<tr>
<td>package CPIMA CharacteristicPolynomialInMonogenicalAlgebra</td>
<td>183</td>
</tr>
<tr>
<td>CharacteristicPolynomialInMonogenicalAlgebra (CPIMA)</td>
<td>184</td>
</tr>
<tr>
<td>package CHARPOL CharacteristicPolynomialPackage</td>
<td>185</td>
</tr>
<tr>
<td>CharacteristicPolynomialPackage (CHARPOL)</td>
<td>186</td>
</tr>
<tr>
<td>package Ibachin ChineseRemainderToolsForIntegralBases</td>
<td>187</td>
</tr>
<tr>
<td>ChineseRemainderToolsForIntegralBases (IBACHIN)</td>
<td>188</td>
</tr>
<tr>
<td>package CVMP CoerceVectorMatrixPackage</td>
<td>192</td>
</tr>
<tr>
<td>CoerceVectorMatrixPackage (CVMP)</td>
<td>193</td>
</tr>
<tr>
<td>package COMBF CombinatorialFunction</td>
<td>194</td>
</tr>
<tr>
<td>CombinatorialFunction (COMBF)</td>
<td>198</td>
</tr>
<tr>
<td>package CDEN CommonDenominator</td>
<td>210</td>
</tr>
<tr>
<td>CommonDenominator (CDEN)</td>
<td>211</td>
</tr>
<tr>
<td>package COMMONOP CommonOperators</td>
<td>212</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
</tr>
<tr>
<td>CommonOperators (COMMONOP)</td>
<td>213</td>
</tr>
<tr>
<td>package COMMUPC CommuteUnivariatePolynomialCategory</td>
<td>218</td>
</tr>
<tr>
<td>CommuteUnivariatePolynomialCategory (COMMUPC)</td>
<td>219</td>
</tr>
<tr>
<td>package COMPFAC ComplexFactorization</td>
<td>220</td>
</tr>
<tr>
<td>ComplexFactorization (COMPFAC)</td>
<td>221</td>
</tr>
<tr>
<td>package COMPLEX2 ComplexFunctions2</td>
<td>223</td>
</tr>
<tr>
<td>ComplexFunctions2 (COMPLEX2)</td>
<td>224</td>
</tr>
<tr>
<td>package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation</td>
<td>225</td>
</tr>
<tr>
<td>ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE)</td>
<td>226</td>
</tr>
<tr>
<td>package COMPLPAT ComplexPattern</td>
<td>227</td>
</tr>
<tr>
<td>ComplexPattern (COMPLPAT)</td>
<td>228</td>
</tr>
<tr>
<td>package CPMATCH ComplexPatternMatch</td>
<td>229</td>
</tr>
<tr>
<td>ComplexPatternMatch (CPMATCH)</td>
<td>230</td>
</tr>
<tr>
<td>package CRFP ComplexRootFindingPackage</td>
<td>231</td>
</tr>
<tr>
<td>ComplexRootFindingPackage (CRFP)</td>
<td>233</td>
</tr>
<tr>
<td>package CMPLXRT ComplexRootPackage</td>
<td>245</td>
</tr>
<tr>
<td>ComplexRootPackage (CMPLXRT)</td>
<td>246</td>
</tr>
<tr>
<td>package CTRIGMNP ComplexTrigonometricManipulations</td>
<td>247</td>
</tr>
<tr>
<td>ComplexTrigonometricManipulations (CTRIGMNP)</td>
<td>248</td>
</tr>
<tr>
<td>package ODECONST ConstantLODE</td>
<td>251</td>
</tr>
<tr>
<td>ConstantLODE (ODECONST)</td>
<td>252</td>
</tr>
<tr>
<td>package COORDSYS CoordinateSystems</td>
<td>254</td>
</tr>
<tr>
<td>CoordinateSystems (COORDSYS)</td>
<td>255</td>
</tr>
<tr>
<td>package CRApackage CRApackage</td>
<td>259</td>
</tr>
<tr>
<td>CRApackage (CRApackage)</td>
<td>260</td>
</tr>
<tr>
<td>package CYCLES CycleIndicators</td>
<td>262</td>
</tr>
<tr>
<td>CycleIndicators (CYCLES)</td>
<td>281</td>
</tr>
<tr>
<td>package CSTTOOLS CyclicStreamTools</td>
<td>286</td>
</tr>
<tr>
<td>CyclicStreamTools (CSTTOOLS)</td>
<td>287</td>
</tr>
<tr>
<td>package CYCLOMPT CyclotomicPolynomialPackage</td>
<td>289</td>
</tr>
<tr>
<td>CyclotomicPolynomialPackage (CYCLOMPT)</td>
<td>290</td>
</tr>
<tr>
<td>5 Chapter D</td>
<td>293</td>
</tr>
<tr>
<td>package DFINTTLS DefiniteIntegrationTools</td>
<td>293</td>
</tr>
<tr>
<td>DefiniteIntegrationTools (DFINTTLS)</td>
<td>294</td>
</tr>
<tr>
<td>package DEGRED DegreeReductionPackage</td>
<td>299</td>
</tr>
<tr>
<td>DegreeReductionPackage (DEGRED)</td>
<td>300</td>
</tr>
<tr>
<td>package DTP DesingTreePackage</td>
<td>302</td>
</tr>
<tr>
<td>DesingTreePackage (DTP)</td>
<td>303</td>
</tr>
<tr>
<td>package DIOSP DiophantineSolutionPackage</td>
<td>312</td>
</tr>
<tr>
<td>DiophantineSolutionPackage (DIOSP)</td>
<td>313</td>
</tr>
<tr>
<td>package DIRPROD2 DirectProductFunctions2</td>
<td>317</td>
</tr>
<tr>
<td>DirectProductFunctions2 (DIRPROD2)</td>
<td>318</td>
</tr>
<tr>
<td>package DLP DiscreteLogarithmPackage</td>
<td>320</td>
</tr>
<tr>
<td>DiscreteLogarithmPackage (DLP)</td>
<td>321</td>
</tr>
<tr>
<td>package DISPLAY DisplayPackage</td>
<td>323</td>
</tr>
</tbody>
</table>
CONTENTS

DisplayPackage (DISPLAY) ........................................... 324
package DDFACT DistinctDegreeFactorize .................................. 326
DistinctDegreeFactorize (DDFACT) ................................ 327
package DFSFUN DoubleFloatSpecialFunctions .......................... 333
DoubleFloatSpecialFunctions (DFSFUN) .................................. 348
The Exponential Integral ............................................... 352
En:(PI,R)→OPR ....................................................... 358
The Ei Function ....................................................... 358
The Fresnel Integral [?, ?] ............................................. 385
package DBLRESP DoubleResultantPackage ............................ 389
DoubleResultantPackage (DBLRESP) ................................ 390
package DRAWCX DrawComplex .......................................... 391
DrawComplex (DRAWCX) ............................................... 392
package DRAWHACK DrawNumericHack .................................. 396
DrawNumericHack (DRAWHACK) ........................................ 397
package DROPT0 DrawOptionFunctions0 ................................ 398
DrawOptionFunctions0 (DROPT0) ...................................... 399
package DROPT1 DrawOptionFunctions1 ................................ 403
DrawOptionFunctions1 (DROPT1) ...................................... 404
package D01AGNT d01AgentsPackage .................................... 405
d01AgentsPackage (D01AGNT) ....................................... 406
package D01WGTS d01WeightsPackage .................................. 412
d01WeightsPackage (D01WGTS) ...................................... 413
package D02AGNT d02AgentsPackage .................................... 419
d02AgentsPackage (D02AGNT) ....................................... 420
package D03AGNT d03AgentsPackage .................................... 426
d03AgentsPackage (D03AGNT) ....................................... 427

6 Chapter E ............................................................. 431
package EP EigenPackage .................................................. 431
EigenPackage (EP) ...................................................... 432
package EF ElementaryFunction ........................................ 437
ElementaryFunction (EF) ............................................... 449
package DEFINTEF ElementaryFunctionDefiniteIntegration ............ 467
ElementaryFunctionDefiniteIntegration (DEFINTEF) .................... 468
package LODEEF ElementaryFunctionLODESolver ........................ 473
ElementaryFunctionLODESolver (LODEEF) ............................... 474
package ODEEF ElementaryFunctionODESolver ............................ 480
ElementaryFunctionODESolver (ODEEF) ................................ 481
package SIGNEF ElementaryFunctionSign ................................ 486
ElementaryFunctionSign (SIGNEF) ..................................... 487
package EFSTRUC ElementaryFunctionStructurePackage ............... 492
ElementaryFunctionStructurePackage (EFSTRUC) ......................... 493
package INTEF ElementaryIntegration ................................... 501
ElementaryIntegration (INTEF) ........................................ 502
package RDEEF ElementaryRischDE ....................................... 511
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementaryRischDE (RDEEF)</td>
<td>512</td>
</tr>
<tr>
<td>package RDEEFS ElementaryRischDESystem</td>
<td>520</td>
</tr>
<tr>
<td>ElementaryRischDESystem (RDEEFS)</td>
<td>521</td>
</tr>
<tr>
<td>package ELFUTS EllipticFunctionsUnivariateTaylorSeries</td>
<td>523</td>
</tr>
<tr>
<td>EllipticFunctionsUnivariateTaylorSeries (ELFUTS)</td>
<td>524</td>
</tr>
<tr>
<td>package EQ2 EquationFunctions2</td>
<td>526</td>
</tr>
<tr>
<td>EquationFunctions2 (EQ2)</td>
<td>527</td>
</tr>
<tr>
<td>package ERROR ErrorFunctions</td>
<td>527</td>
</tr>
<tr>
<td>ErrorFunctions (ERROR)</td>
<td>529</td>
</tr>
<tr>
<td>package GBEUCLID EuclideanGroebnerBasisPackage</td>
<td>531</td>
</tr>
<tr>
<td>EuclideanGroebnerBasisPackage (GBEUCLID)</td>
<td>555</td>
</tr>
<tr>
<td>package EVALCYC EvaluateCycleIndicators</td>
<td>566</td>
</tr>
<tr>
<td>EvaluateCycleIndicators (EVALCYC)</td>
<td>567</td>
</tr>
<tr>
<td>package ESCONT ExpertSystemContinuityPackage</td>
<td>568</td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage (ESCONT)</td>
<td>569</td>
</tr>
<tr>
<td>package ESCONT1 ExpertSystemContinuityPackage1</td>
<td>575</td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage1 (ESCONT1)</td>
<td>576</td>
</tr>
<tr>
<td>package ESTOOLS ExpertSystemToolsPackage</td>
<td>577</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage (ESTOOLS)</td>
<td>578</td>
</tr>
<tr>
<td>package ESTOOLS1 ExpertSystemToolsPackage1</td>
<td>586</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage1 (ESTOOLS1)</td>
<td>587</td>
</tr>
<tr>
<td>package ESTOOLS2 ExpertSystemToolsPackage2</td>
<td>588</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage2 (ESTOOLS2)</td>
<td>589</td>
</tr>
<tr>
<td>package EXPR2 ExpressionFunctions2</td>
<td>590</td>
</tr>
<tr>
<td>ExpressionFunctions2 (EXPR2)</td>
<td>591</td>
</tr>
<tr>
<td>package EXPRSOL ExpressionSolve</td>
<td>592</td>
</tr>
<tr>
<td>Bugs</td>
<td>592</td>
</tr>
<tr>
<td>ExpressionSolve (EXPRSOL)</td>
<td>593</td>
</tr>
<tr>
<td>package ES1 ExpressionSpaceFunctions1</td>
<td>595</td>
</tr>
<tr>
<td>ExpressionSpaceFunctions1 (ES1)</td>
<td>596</td>
</tr>
<tr>
<td>package ES2 ExpressionSpaceFunctions2</td>
<td>597</td>
</tr>
<tr>
<td>ExpressionSpaceFunctions2 (ES2)</td>
<td>598</td>
</tr>
<tr>
<td>package EXPRODE ExpressionSpaceODESolver</td>
<td>599</td>
</tr>
<tr>
<td>ExpressionSpaceODESolver (EXPRODE)</td>
<td>600</td>
</tr>
<tr>
<td>package OMEXPR ExpressionToOpenMath</td>
<td>605</td>
</tr>
<tr>
<td>ExpressionToOpenMath (OMEXPR)</td>
<td>606</td>
</tr>
<tr>
<td>package EXPR2UPS ExpressionToUnivariatePowerSeries</td>
<td>612</td>
</tr>
<tr>
<td>ExpressionToUnivariatePowerSeries (EXPR2UPS)</td>
<td>613</td>
</tr>
<tr>
<td>package EXPRTUBE ExpressionTubePlot</td>
<td>620</td>
</tr>
<tr>
<td>ExpressionTubePlot (EXPRTUBE)</td>
<td>621</td>
</tr>
<tr>
<td>package EXP3D Export3D</td>
<td>625</td>
</tr>
<tr>
<td>Export3D (EXP3D)</td>
<td>626</td>
</tr>
<tr>
<td>package E04AGNT e04AgentsPackage</td>
<td>628</td>
</tr>
<tr>
<td>e04AgentsPackage (E04AGNT)</td>
<td>629</td>
</tr>
</tbody>
</table>
CONTENTS

FloatingComplexPackage (FLOATCP) ........................................ 735
package FLOATRP FloatingRealPackage ................................ 738
FloatingRealPackage (FLOATRP) ............................................ 739
package FCPAK1 FortranCodePackage1 .................................... 742
FortranCodePackage1 (FCPAK1) ............................................ 743
package FOP FortranOutputStackPackage .................................. 746
FortranOutputStackPackage (FOP) ......................................... 747
package FORT FortranPackage ................................................ 749
FortranPackage (FORT) ....................................................... 750
package FRIDEAL2 FractionalIdealFunctions2 ............................ 752
FractionalIdealFunctions2 (FRIDEAL2) .................................... 753
package FFFG FractionFreeFastGaussian .................................. 754
FractionFreeFastGaussian (FFFG) .......................................... 756
package FFFGF FractionFreeFastGaussianFractions ........................ 766
FractionFreeFastGaussianFractions (FFFGF) ............................... 767
package FRAC2 FractionFunctions2 ................................ ........ 770
FractionFunctions2 (FRAC2) ................................................. 771
package FRNAAF2 FramedNonAssociativeAlgebraFunctions2 .......... 772
FramedNonAssociativeAlgebraFunctions2 (FRNAAF2) .................. 773
package FSPECF FunctionalSpecialFunction ............................. 774
FunctionalSpecialFunction (FSPECF) ...................................... 775
differentiation of special functions ....................................... 781
package FFCAT2 FunctionFieldCategoryFunctions2 ........................ 783
FunctionFieldCategoryFunctions2 (FFCAT2) ............................... 784
package FFINTBAS FunctionFieldIntegralBasis ........................... 785
FunctionFieldIntegralBasis (FFINTBAS) .................................. 786
package PMASSFS FunctionSpaceAssertions ................................ 789
FunctionSpaceAssertions (PMASSFS) ...................................... 790
package PMPREDFS FunctionSpaceAttachPredicates ........................ 792
FunctionSpaceAttachPredicates (PMPREDFS) ............................. 793
package FSCINT FunctionSpaceComplexIntegration ........................ 795
FunctionSpaceComplexIntegration (FSCINT) ............................... 796
package FS2 FunctionSpaceFunctions2 .................................... 798
FunctionSpaceFunctions2 (FS2) ............................................. 799
package FSINT FunctionSpaceIntegration .................................. 800
FunctionSpaceIntegration (FSINT) ......................................... 801
package FSPRMELT FunctionSpacePrimitiveElement ........................ 804
FunctionSpacePrimitiveElement (FSPRMELT) .............................. 805
package FSRED FunctionSpaceReduce ...................................... 808
FunctionSpaceReduce (FSRED) ............................................. 809
package SUMFS FunctionSpaceSum ......................................... 810
FunctionSpaceSum (SUMFS) .................................................. 811
package FS2EXPXP FunctionSpaceToExponentialExpansion .............. 813
FunctionSpaceToExponentialExpansion (FS2EXPXP) .................... 814
package FS2UPS FunctionSpaceToUnivariatePowerSeries ................ 825
FunctionSpaceToUnivariatePowerSeries (FS2UPS) ...................... 826
<table>
<thead>
<tr>
<th>Package Name</th>
<th>842</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionSpaceUnivariatePolynomialFactor</td>
<td></td>
</tr>
<tr>
<td>FunctionSpaceUnivariatePolynomialFactor (FSUPFACT)</td>
<td>843</td>
</tr>
</tbody>
</table>

8 Chapter G 847

<table>
<thead>
<tr>
<th>Package Name</th>
<th>847</th>
</tr>
</thead>
<tbody>
<tr>
<td>GaloisGroupFactorizationUtilities</td>
<td></td>
</tr>
<tr>
<td>GaloisGroupFactorizationUtilities (GALFACTU)</td>
<td>847</td>
</tr>
<tr>
<td>GaloisGroupFactorizer</td>
<td>851</td>
</tr>
<tr>
<td>GaloisGroupFactorizer (GALFACT)</td>
<td>852</td>
</tr>
<tr>
<td>GaloisGroupPolynomialUtilities</td>
<td>869</td>
</tr>
<tr>
<td>GaloisGroupPolynomialUtilities (GALPOLYU)</td>
<td>870</td>
</tr>
<tr>
<td>GaloisGroupUtilities</td>
<td>872</td>
</tr>
<tr>
<td>GaloisGroupUtilities (GALUTIL)</td>
<td>873</td>
</tr>
<tr>
<td>GaussianFactorizationPackage</td>
<td>876</td>
</tr>
<tr>
<td>GaussianFactorizationPackage (GAUSSFAC)</td>
<td>877</td>
</tr>
<tr>
<td>GeneralHenselPackage</td>
<td>881</td>
</tr>
<tr>
<td>GeneralHenselPackage (GHENSEL)</td>
<td>882</td>
</tr>
<tr>
<td>GeneralizedMultivariateFactorize</td>
<td>885</td>
</tr>
<tr>
<td>GeneralizedMultivariateFactorize (GENMFACT)</td>
<td>886</td>
</tr>
<tr>
<td>GeneralPackageForAlgebraicFunctionField</td>
<td>887</td>
</tr>
<tr>
<td>GeneralPackageForAlgebraicFunctionField (GPAFF)</td>
<td>889</td>
</tr>
<tr>
<td>GeneralPolynomialGcdPackage</td>
<td>903</td>
</tr>
<tr>
<td>GeneralPolynomialGcdPackage (GENPGCD)</td>
<td>904</td>
</tr>
<tr>
<td>GenerateUnivariatePowerSeries</td>
<td>917</td>
</tr>
<tr>
<td>GenerateUnivariatePowerSeries (GENUPS)</td>
<td>918</td>
</tr>
<tr>
<td>GenExEuclid (GENEEZ)</td>
<td>923</td>
</tr>
<tr>
<td>GenExEuclid (GENEEZ)</td>
<td>924</td>
</tr>
<tr>
<td>GenUFactorize (GENUFAC)</td>
<td>928</td>
</tr>
<tr>
<td>GenUFactorize (GENUFAC)</td>
<td>929</td>
</tr>
<tr>
<td>GenusZeroIntegration</td>
<td>931</td>
</tr>
<tr>
<td>GenusZeroIntegration (INTG0)</td>
<td>932</td>
</tr>
<tr>
<td>GnuDraw (GDRAW)</td>
<td>938</td>
</tr>
<tr>
<td>GnuDraw (GDRAW)</td>
<td>939</td>
</tr>
<tr>
<td>GosperSummationMethod</td>
<td>941</td>
</tr>
<tr>
<td>GosperSummationMethod (GOSPER)</td>
<td>942</td>
</tr>
<tr>
<td>GraphicsDefaults</td>
<td>947</td>
</tr>
<tr>
<td>GraphicsDefaults (GRDEF)</td>
<td>948</td>
</tr>
<tr>
<td>GrayCode (GRAY)</td>
<td>950</td>
</tr>
<tr>
<td>GrayCode (GRAY)</td>
<td>951</td>
</tr>
<tr>
<td>GroebnerFactorizationPackage</td>
<td>953</td>
</tr>
<tr>
<td>GroebnerFactorizationPackage (GBF)</td>
<td>957</td>
</tr>
<tr>
<td>GroebnerInternalPackage</td>
<td>964</td>
</tr>
<tr>
<td>GroebnerInternalPackage (GBINTERN)</td>
<td>965</td>
</tr>
<tr>
<td>GroebnerPackage</td>
<td>975</td>
</tr>
<tr>
<td>GroebnerPackage (GB)</td>
<td>1002</td>
</tr>
<tr>
<td>GroebnerSolve</td>
<td>1005</td>
</tr>
<tr>
<td>GroebnerSolve (GROEBSOL)</td>
<td>1006</td>
</tr>
</tbody>
</table>
| CONTENTS | xi
|---|---
| package GUESS Guess | 1011 
| Guess (GUESS) | 1012 
| package GUESSAN GuessAlgebraicNumber | 1047 
| GuessAlgebraicNumber (GUESSAN) | 1048 
| package GUESSF GuessFinite | 1049 
| GuessFinite (GUESSF) | 1050 
| package GUESSF1 GuessFiniteFunctions | 1050 
| GuessFiniteFunctions (GUESSF1) | 1051 
| package GUESSINT GuessInteger | 1052 
| GuessInteger (GUESSINT) | 1053 
| package GUESSP GuessPolynomial | 1054 
| GuessPolynomial (GUESSP) | 1055 
| package GUESSUP GuessUnivariatePolynomial | 1056 
| GuessUnivariatePolynomial (GUESSUP) | 1057 

9 Chapter H 
| package HB HallBasis | 1063 
| HallBasis (HB) | 1064 
| package HEUGCD HenGcd | 1066 
| HenGcd (HEUGCD) | 1067 

10 Chapter I 
| package IDECOMP IdealDecompositionPackage | 1073 
| IdealDecompositionPackage (IDECOMP) | 1074 
| package INCRMAPS IncrementingMaps | 1082 
| IncrementingMaps (INCRMAPS) | 1083 
| package INFPROD0 InfiniteProductCharacteristicZero | 1084 
| InfiniteProductCharacteristicZero (INFPROD0) | 1085 
| package INFPRODF FiniteProductFiniteField | 1086 
| InfiniteProductFiniteField (INFPRODF) | 1087 
| package INFRODPF InfiniteProductPrimeField | 1090 
| InfiniteProductPrimeField (INFRODPF) | 1091 
| package ITFUN2 InfiniteTupleFunctions2 | 1092 
| InfiniteTupleFunctions2 (ITFUN2) | 1093 
| package ITFUN3 InfiniteTupleFunctions3 | 1094 
| InfiniteTupleFunctions3 (ITFUN3) | 1095 
| package INFINITY Infinity | 1096 
| Infinity (INFINITY) | 1097 
| package IALGFACT InnerAlgFactor | 1098 
| InnerAlgFactor (IALGFACT) | 1099 
| package ICDEN InnerCommonDenominator | 1101 
| InnerCommonDenominator (ICDEN) | 1102 
| package IMATLIN InnerMatrixLinearAlgebraFunctions | 1104 
| InnerMatrixLinearAlgebraFunctions (IMATLIN) | 1105 
| package IMATQF InnerMatrixQuotientFieldFunctions | 1110 
| InnerMatrixQuotientFieldFunctions (IMATQF) | 1111 |
package INMODGCD InnerModularGcd ........................................... . 1112
  InnerModularGcd (INMODGCD) .......................................... . 1113
package INNMFACT InnerMultFact ........................................... . 1119
  InnerMultFact (INNMFACT) .............................................. . 1120
package INBFF InnerNormalBasisFieldFunctions .......................... . 1129
  InnerNormalBasisFieldFunctions (INBFF) ................................ . 1130
package INEP InnerNumericEigenPackage .................................. . 1137
  InnerNumericEigenPackage (INEP) ...................................... . 1138
package INFSP InnerNumericFloatSolvePackage ............................ . 1143
  InnerNumericFloatSolvePackage (INFSP) ................................ . 1144
package INPSIGN InnerPolySign ........................................... . 1148
  InnerPolySign (INPSIGN) ............................................. . 1149
package ISUMP InnerPolySum ............................................. . 1150
  InnerPolySum (ISUMP) ................................................ . 1151
package ITRIGMNP InnerTrigonometricManipulations ........................ . 1153
  InnerTrigonometricManipulations (ITRIGMNP) ............................ . 1154
package INFORM1 InputFormFunctions1 .................................... . 1158
  InputFormFunctions1 (INFORM1) ....................................... . 1159
package INTERGB InterfaceGroebnerPackage ................................ . 1160
  InterfaceGroebnerPackage (INTERGB) .................................. . 1161
package ZLINDEP IntegerLinearDependence ................................ . 1179
  IntegerLinearDependence (ZLINDEP) ................................... . 1182
package INTHEORY IntegerNumberTheoryFunctions .......................... . 1183
  IntegerNumberTheoryFunctions (INTHEORY) .............................. . 1197
package PRIMES IntegerPrimesPackage .................................... . 1202
  IntegerPrimesPackage (PRIMES) ...................................... . 1203
package IROOT IntegerRoots .............................................. . 1215
  IntegerRoots (IROOT) ................................................ . 1215
CONTENTS

IntegerRoots (IROOT) ........................................ 1216
perfectSquare? ................................. 1217
perfectNthPower? ......................... 1217
perfectNthRoot ............................. 1218
approxNthRoot ............................... 1218
perfectNthRoot ............................. 1219
perfectSqrt ................................... 1219
approxSqrt ................................... 1219
package INTSLPE IntegerSolveLinearPolynomialEquation .................. 1220
IntegerSolveLinearPolynomialEquation (INTSLPE) ....................... 1221
package IBATOOL IntegralBasisTools .................................. 1222
IntegralBasisTools (IBATOOL) ...................... 1223
package IBPTOOLS IntegralBasisPolynomialTools .......................... 1226
IntegralBasisPolynomialTools (IBPTOOLS) ......................... 1227
package IR2 IntegrationResultFunctions2 ................................ 1229
IntegrationResultFunctions2 (IR2) ...................... 1230
package IRRF2F IntegrationResultRFToFunction ......................... 1232
IntegrationResultRFToFunction (IRRF2F) ...................... 1233
package IR2F IntegrationResultToFunction ............................ 1234
IntegrationResultToFunction (IR2F) ...................... 1236
package INTTOOLS IntegrationTools .................................... 1241
IntegrationTools (INTTOOLS) ...................... 1242
package IPRNTPK InternalPrintPackage ........................................ 1245
InternalPrintPackage (IPRNTPK) ...................... 1246
package IRURPK InternalRationalUnivariateRepresentationPackage ........ 1247
InternalRationalUnivariateRepresentationPackage (IRURPK) .......... 1248
package INTFRSP InterpolateFormsPackage ................................ 1252
InterpolateFormsPackage (INTFRSP) ...................... 1253
package INTDIVP IntersectionDivisorPackage ............................ 1260
IntersectionDivisorPackage (INTDIVP) ...................... 1261
package IRREDFFX IrredPolyOverFiniteField ............................. 1263
IrredPolyOverFiniteField (IRREDFFX) ...................... 1264
package IRSN IrrRepSymNatPackage ..................................... 1266
IrrRepSymNatPackage (IRSN) ...................... 1267
package INVLAPLA InverseLaplaceTransform ............................... 1274
InverseLaplaceTransform (INVLAPLA) ...................... 1275

11 Chapter J ........................................ 1279

12 Chapter K ........................................ 1281
package KERNEL2 KernelFunctions2 .................................... 1281
KernelFunctions2 (KERNEL2) ...................... 1282
package KOVACIC Kovacic ............................................. 1283
Kovacic (KOVACIC) ................................. 1284
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Package Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>LAPLACE</td>
<td>LaplaceTransform</td>
<td>1287</td>
</tr>
<tr>
<td></td>
<td>(LAPLACE)</td>
<td></td>
<td>1288</td>
</tr>
<tr>
<td></td>
<td>LAZM3PK</td>
<td>LazardSetSolvingPackage</td>
<td>1293</td>
</tr>
<tr>
<td></td>
<td>(LAZM3PK)</td>
<td></td>
<td>1313</td>
</tr>
<tr>
<td></td>
<td>LEADCDET</td>
<td>LeadingCoeffDetermination</td>
<td>1316</td>
</tr>
<tr>
<td></td>
<td>(LEADCDET)</td>
<td></td>
<td>1317</td>
</tr>
<tr>
<td></td>
<td>LEXTRIPK</td>
<td>LexTriangularPackage</td>
<td>1320</td>
</tr>
<tr>
<td></td>
<td>(LEXTRIPK)</td>
<td></td>
<td>1390</td>
</tr>
<tr>
<td></td>
<td>LINDEP</td>
<td>LinearDependence</td>
<td>1395</td>
</tr>
<tr>
<td></td>
<td>(LINDEP)</td>
<td></td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td>LODOF</td>
<td>LinearOrdinaryDifferentialOperatorFactorizer</td>
<td>1398</td>
</tr>
<tr>
<td></td>
<td>(LODOF)</td>
<td></td>
<td>1399</td>
</tr>
<tr>
<td></td>
<td>LODOOPS</td>
<td>LinearOrdinaryDifferentialOperatorsOps</td>
<td>1402</td>
</tr>
<tr>
<td></td>
<td>(LODOOPS)</td>
<td></td>
<td>1403</td>
</tr>
<tr>
<td></td>
<td>LPEFRAC</td>
<td>LinearPolynomialEquationByFractions</td>
<td>1405</td>
</tr>
<tr>
<td></td>
<td>(LPEFRAC)</td>
<td></td>
<td>1406</td>
</tr>
<tr>
<td></td>
<td>LISYSER</td>
<td>LinearSystemFromPowerSeriesPackage</td>
<td>1408</td>
</tr>
<tr>
<td></td>
<td>(LISYSER)</td>
<td></td>
<td>1409</td>
</tr>
<tr>
<td></td>
<td>LMP</td>
<td>LinearSystemMatrixPackage</td>
<td>1411</td>
</tr>
<tr>
<td></td>
<td>(LMP)</td>
<td></td>
<td>1412</td>
</tr>
<tr>
<td></td>
<td>LMP1</td>
<td>LinearSystemMatrixPackage1</td>
<td>1414</td>
</tr>
<tr>
<td></td>
<td>(LMP1)</td>
<td></td>
<td>1415</td>
</tr>
<tr>
<td></td>
<td>LSPP</td>
<td>LinearSystemPolynomialPackage</td>
<td>1417</td>
</tr>
<tr>
<td></td>
<td>(LSPP)</td>
<td></td>
<td>1418</td>
</tr>
<tr>
<td></td>
<td>LGROBP</td>
<td>LinGroebnerPackage</td>
<td>1419</td>
</tr>
<tr>
<td></td>
<td>(LGROBP)</td>
<td></td>
<td>1420</td>
</tr>
<tr>
<td></td>
<td>LOP</td>
<td>LinesOpPack</td>
<td>1427</td>
</tr>
<tr>
<td></td>
<td>(LOP)</td>
<td></td>
<td>1428</td>
</tr>
<tr>
<td></td>
<td>LF</td>
<td>LiouvillianFunction</td>
<td>1431</td>
</tr>
<tr>
<td></td>
<td>(LF)</td>
<td></td>
<td>1432</td>
</tr>
<tr>
<td></td>
<td>LIST2</td>
<td>ListFunctions2</td>
<td>1436</td>
</tr>
<tr>
<td></td>
<td>(LIST2)</td>
<td></td>
<td>1437</td>
</tr>
<tr>
<td></td>
<td>LIST3</td>
<td>ListFunctions3</td>
<td>1439</td>
</tr>
<tr>
<td></td>
<td>(LIST3)</td>
<td></td>
<td>1440</td>
</tr>
<tr>
<td></td>
<td>LIST2MAP</td>
<td>ListToMap</td>
<td>1441</td>
</tr>
<tr>
<td></td>
<td>(LIST2MAP)</td>
<td></td>
<td>1442</td>
</tr>
<tr>
<td></td>
<td>LPARSPT</td>
<td>LocalParametrizationOfSimplePointPackage</td>
<td>1444</td>
</tr>
<tr>
<td></td>
<td>(LPARSPT)</td>
<td></td>
<td>1445</td>
</tr>
<tr>
<td>M</td>
<td>MKBCFUNC</td>
<td>MakeBinaryCompiledFunction</td>
<td>1453</td>
</tr>
<tr>
<td></td>
<td>(MKBCFUNC)</td>
<td></td>
<td>1454</td>
</tr>
<tr>
<td></td>
<td>MKFLCFN</td>
<td>MakeFloatCompiledFunction</td>
<td>1455</td>
</tr>
<tr>
<td></td>
<td>(MKFLCFN)</td>
<td></td>
<td>1456</td>
</tr>
<tr>
<td></td>
<td>MKFUNC</td>
<td>MakeFunction</td>
<td>1459</td>
</tr>
</tbody>
</table>
CONTENTS

MakeFunction (MKFUNC) ................................................. 1464
package MKRECORD MakeRecord ..................................... 1465
    MakeRecord (MKRECORD) ..................................... 1466
package MKUCFUNC MakeUnaryCompiledFunction .................. 1467
    MakeUnaryCompiledFunction (MKUCFUNC) ..................... 1468
package MAPHACK1 MappingPackageInternalHacks1 ............. 1469
    MappingPackageInternalHacks1 (MAPHACK1) ................. 1470
package MAPHACK2 MappingPackageInternalHacks2 ............. 1471
    MappingPackageInternalHacks2 (MAPHACK2) ................. 1472
package MAPHACK3 MappingPackageInternalHacks3 ............. 1473
    MappingPackageInternalHacks3 (MAPHACK3) ................. 1474
package MAPPKG1 MappingPackage1 .................................. 1475
    MappingPackage1 (MAPPKG1) ................................ 1484
package MAPPKG2 MappingPackage2 .................................. 1486
    MappingPackage2 (MAPPKG2) ................................ 1494
package MAPPKG3 MappingPackage3 .................................. 1496
package MAPPKG4 MappingPackage4 .................................. 1505
package MATCAT2 MatrixCategoryFunctions2 .................... 1514
    MatrixCategoryFunctions2 (MATCAT2) ....................... 1515
package MCDEN MatrixCommonDenominator ....................... 1516
    MatrixCommonDenominator (MCDEN) .......................... 1517
package MATLIN MatrixLinearAlgebraFunctions .................. 1519
    MatrixLinearAlgebraFunctions (MATLIN) .................... 1520
package MAMA MatrixManipulation .................................. 1527
    MatrixManipulation (MAMA) .................................. 1542
package MTHING MergeThing ........................................ 1549
    MergeThing (MTHING) ........................................ 1550
package MESH MeshCreationRoutinesForThreeDimensions ....... 1551
    MeshCreationRoutinesForThreeDimensions (MESH) .......... 1552
package MDDFACT ModularDistinctDegreeFactorizer ............. 1555
    ModularDistinctDegreeFactorizer (MDDFACT) ............... 1556
package MHROWRED ModularHermitianRowReduction ............... 1561
    ModularHermitianRowReduction (MHROWRED) .................. 1562
package MRF2 MonoidRingFunctions2 ................................ 1567
    MonoidRingFunctions2 (MRF2) ................................ 1568
package MONOTOOL MonomialExtensionTools ...................... 1569
    MonomialExtensionTools (MONOTOOL) ....................... 1570
package MSYSCMD MoreSystemCommands ............................ 1572
    MoreSystemCommands (MSYSCMD) ............................. 1573
package MPCPF MPolyCatPolyFactorizer ........................... 1574
    MPolyCatPolyFactorizer (MPCPF) ............................ 1575
package MPRFF MPolyCatRationalFunctionFactorizer ............. 1577
    MPolyCatRationalFunctionFactorizer (MPRFF) .............. 1578
package MPC2 MPolyCatFunctions2 ................................ 1581
MPolyCatFunctions2 (MPC2) .............................................. 1582
package MPC3 MPolyCatFunctions3 .............................................. 1583
MPolyCatFunctions3 (MPC3) ...................................................... 1584
package MRATFAC MRationalFactorize ............................................ 1585
MRationalFactorize (MRATFAC) .............................................. 1586
package MFINFACT MultFiniteFactorize ........................................ 1588
MultFiniteFactorize (MFINFACT) .............................................. 1589
package MMAP MultipleMap .................................................... 1599
MultipleMap (MMAP) ....................................................... 1600
package MCALCFN MultiVariableCalculusFunctions ......................... 1601
MultiVariableCalculusFunctions (MCALCFN) .................................. 1602
package MULTFACT MultivariateFactorize ..................................... 1606
MultivariateFactorize (MULTFACT) .............................................. 1607
packageMLIFT MultivariateLifting ............................................. 1608
package MULTSQFR MultivariateSquareFree .................................. 1614
MultivariateSquareFree (MULTSQFR) ........................................... 1615

15 Chapter N ........................................................................ 1623
package NAGF02 NagEigenPackage ............................................ 1623
NagEigenPackage (NAGF02) ..................................................... 1690
package NAGE02 NagFittingPackage ............................................ 1702
NagFittingPackage (NAGE02) ................................................... 1832
package NAGF04 NagLinearEquationSolvingPackage ......................... 1845
NagLinearEquationSolvingPackage (NAGF04) .................................. 1910
package NAGSP NAGLinkSupportPackage .................................... 1919
NAGLinkSupportPackage (NAGSP) .............................................. 1920
package NAGD01 NagIntegrationPackage ..................................... 1922
NagIntegrationPackage (NAGD01) .............................................. 1999
package NAGE01 NagInterpolationPackage .................................. 2008
NagInterpolationPackage (NAGE01) ............................................. 2047
package NAGF07 NagLapack ..................................................... 2054
NagLapack (NAGF07) ............................................................. 2068
package NAGF01 NagMatrixOperationsPackage ................................ 2071
NagMatrixOperationsPackage (NAGF01) ....................................... 2128
package NAGE04 NagOptimisationPackage ................................... 2135
NagOptimisationPackage (NAGE04) .............................................. 2287
package NAGD02 NagOrdinaryDifferentialEquationsPackage ................ 2296
NagOrdinaryDifferentialEquationsPackage (NAGD02) ......................... 2386
package NAGD03 NagPartialDifferentialEquationsPackage ................. 2396
NagPartialDifferentialEquationsPackage (NAGD03) ......................... 2433
package NAGC02 NagPolynomialRootsPackage ................................ 2436
NagPolynomialRootsPackage (NAGC02) ........................................... 2451
package NAGC05 NagRootFindingPackage .................................... 2453
NagRootFindingPackage (NAGC05) .............................................. 2470
package NAGC06 NagSeriesSummationPackage ................................ 2473
NagSeriesSummationPackage (NAGC06) ........................................... 2518
### CONTENTS

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>package NAGS NagSpecialFunctionsPackage</td>
<td>NagSpecialFunctionsPackage (NAGS)</td>
<td>2524</td>
</tr>
<tr>
<td>package NSUP2 NewSparseUnivariatePolynomialFunctions2</td>
<td>NewSparseUnivariatePolynomialFunctions2 (NSUP2)</td>
<td>2687</td>
</tr>
<tr>
<td>package NEWTON NewtonInterpolation</td>
<td>NewtonInterpolation (NEWTON)</td>
<td>2689</td>
</tr>
<tr>
<td>package NPOLYON NewtonPolygon</td>
<td>NewtonPolygon (NPOLYON)</td>
<td>2690</td>
</tr>
<tr>
<td>package NCODIV NonCommutativeOperatorDivision</td>
<td>NonCommutativeOperatorDivision (NCODIV)</td>
<td>2697</td>
</tr>
<tr>
<td>package NONE1 NoneFunctions1</td>
<td>NoneFunctions1 (NONE1)</td>
<td>2700</td>
</tr>
<tr>
<td>package NODE1 NonLinearFirstOrderODESolver</td>
<td>NonLinearFirstOrderODESolver (NODE1)</td>
<td>2702</td>
</tr>
<tr>
<td>package NLINSOL NonLinearSolvePackage</td>
<td>NonLinearSolvePackage (NLINSOL)</td>
<td>2706</td>
</tr>
<tr>
<td>package NORMPK NormalizationPackage</td>
<td>NormalizationPackage (NORMPK)</td>
<td>2710</td>
</tr>
<tr>
<td>package NORMMA NormInMonogenicAlgebra</td>
<td>NormInMonogenicAlgebra (NORMMA)</td>
<td>2715</td>
</tr>
<tr>
<td>package NORMRETR NormRetractPackage</td>
<td>NormRetractPackage (NORMRETR)</td>
<td>2718</td>
</tr>
<tr>
<td>package NPCOEF NPCoef</td>
<td>NPCoef (NPCOEF)</td>
<td>2720</td>
</tr>
<tr>
<td>package NFINTBAS NumberFieldIntegralBasis</td>
<td>NumberFieldIntegralBasis (NFINTBAS)</td>
<td>2725</td>
</tr>
<tr>
<td>package NUMFMT NumberFormats</td>
<td>NumberFormats (NUMFMT)</td>
<td>2731</td>
</tr>
<tr>
<td>package NTPOLFN NumberTheoreticPolynomialFunctions</td>
<td>NumberTheoreticPolynomialFunctions (NTPOLFN)</td>
<td>2736</td>
</tr>
<tr>
<td>package NUMERE Numeric</td>
<td>Numeric (NUMERIC)</td>
<td>2739</td>
</tr>
<tr>
<td>package NUMODE NumericalOrdinaryDifferentialEquations</td>
<td>NumericalOrdinaryDifferentialEquations (NUMODE)</td>
<td>2749</td>
</tr>
<tr>
<td>package NUMQUAD NumericalQuadrature</td>
<td>NumericalQuadrature (NUMQUAD)</td>
<td>2760</td>
</tr>
<tr>
<td>package NCEP NumericComplexEigenPackage</td>
<td>NumericComplexEigenPackage (NCEP)</td>
<td>2773</td>
</tr>
<tr>
<td>package NCNTFRAC NumericContinuedFraction</td>
<td>NumericContinuedFraction (NCNTFRAC)</td>
<td>2776</td>
</tr>
<tr>
<td>package NREP NumericRealEigenPackage</td>
<td>NumericRealEigenPackage (NREP)</td>
<td>2778</td>
</tr>
<tr>
<td>package NUMTUBE NumericTubePlot</td>
<td>NumericTubePlot (NUMTUBE)</td>
<td>2781</td>
</tr>
</tbody>
</table>

---

**Note:** Page numbers are provided for reference.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Package Name</th>
<th>Documentation Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>OCTCT2 OctonionCategoryFunctions2</td>
<td>2787</td>
</tr>
<tr>
<td></td>
<td>ODEINT ODEIntegration</td>
<td>2789</td>
</tr>
<tr>
<td></td>
<td>ODETools (ODETOOLS)</td>
<td>2792</td>
</tr>
<tr>
<td></td>
<td>ARRAY12 OneDimensionalArrayFunctions2</td>
<td>2795</td>
</tr>
<tr>
<td></td>
<td>ODETOOLS ODETools</td>
<td>2793</td>
</tr>
<tr>
<td></td>
<td>ONECOMP2 OnePointCompletionFunctions2</td>
<td>2798</td>
</tr>
<tr>
<td></td>
<td>PACK PACKAGE</td>
<td>2796</td>
</tr>
<tr>
<td></td>
<td>ORDCOMP2 OrderedCompletionFunctions2</td>
<td>2807</td>
</tr>
<tr>
<td></td>
<td>ORDERFUNCTIONS</td>
<td>2808</td>
</tr>
<tr>
<td></td>
<td>ORTHPOL OrthogonalPolynomialFunctions</td>
<td>2812</td>
</tr>
<tr>
<td></td>
<td>OutputPackage (OUT)</td>
<td>2815</td>
</tr>
<tr>
<td></td>
<td>OutputPackage (OUT)</td>
<td>2816</td>
</tr>
<tr>
<td>P</td>
<td>PAFF PackageForAlgebraicFunctionField</td>
<td>2819</td>
</tr>
<tr>
<td></td>
<td>PackageForAlgebraicFunctionField (PAFF)</td>
<td>2821</td>
</tr>
<tr>
<td></td>
<td>PAFF PackageForAlgebraicFunctionFieldOverFiniteField</td>
<td>2827</td>
</tr>
<tr>
<td></td>
<td>PackageForAlgebraicFunctionFieldOverFiniteField (PAFF)</td>
<td>2829</td>
</tr>
<tr>
<td></td>
<td>PFORP PackageForPoly</td>
<td>2837</td>
</tr>
<tr>
<td></td>
<td>PackageForPoly (PFORP)</td>
<td>2838</td>
</tr>
<tr>
<td></td>
<td>PADEPAC PadeApproximantPackage</td>
<td>2845</td>
</tr>
<tr>
<td></td>
<td>PadeApproximantPackage (PADEPAC)</td>
<td>2846</td>
</tr>
<tr>
<td></td>
<td>PADE PadeApproximants</td>
<td>2847</td>
</tr>
<tr>
<td></td>
<td>PadeApproximants (PADE)</td>
<td>2848</td>
</tr>
<tr>
<td></td>
<td>PWFFINTB PAdicWildFunctionFieldIntegralBasis</td>
<td>2852</td>
</tr>
<tr>
<td></td>
<td>PAdicWildFunctionFieldIntegralBasis (PWFFINTB)</td>
<td>2853</td>
</tr>
<tr>
<td></td>
<td>YSTREAM ParadoxicalCombinatorsForStreams</td>
<td>2858</td>
</tr>
<tr>
<td></td>
<td>ParadoxicalCombinatorsForStreams (YSTREAM)</td>
<td>2859</td>
</tr>
<tr>
<td></td>
<td>PLEQN ParametricLinearEquations</td>
<td>2860</td>
</tr>
<tr>
<td></td>
<td>ParametricLinearEquations (PLEQN)</td>
<td>2862</td>
</tr>
<tr>
<td></td>
<td>PARPC2 ParametricPlaneCurveFunctions2</td>
<td>2875</td>
</tr>
<tr>
<td></td>
<td>ParametricPlaneCurveFunctions2 (PARPC2)</td>
<td>2876</td>
</tr>
<tr>
<td></td>
<td>PARSC2 ParametricSpaceCurveFunctions2</td>
<td>2877</td>
</tr>
<tr>
<td>Package Name</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>ParametricSpaceCurveFunctions2 (PARSC2)</td>
<td>2878</td>
<td></td>
</tr>
<tr>
<td>package PARSU2 ParametricSurfaceFunctions2</td>
<td>2878</td>
<td></td>
</tr>
<tr>
<td>ParametricSurfaceFunctions2 (PARSU2)</td>
<td>2879</td>
<td></td>
</tr>
<tr>
<td>package PARAMP ParametrizationPackage</td>
<td>2880</td>
<td></td>
</tr>
<tr>
<td>ParametrizationPackage (PARAMP)</td>
<td>2881</td>
<td></td>
</tr>
<tr>
<td>package PFRPAC PartialFractionPackage</td>
<td>2883</td>
<td></td>
</tr>
<tr>
<td>PartialFractionPackage (PFRPAC)</td>
<td>2886</td>
<td></td>
</tr>
<tr>
<td>package PARTPERM PartitionsAndPermutations</td>
<td>2887</td>
<td></td>
</tr>
<tr>
<td>PartitionsAndPermutations (PARTPERM)</td>
<td>2888</td>
<td></td>
</tr>
<tr>
<td>package PATTERN1 PatternFunctions1</td>
<td>2891</td>
<td></td>
</tr>
<tr>
<td>PatternFunctions1 (PATTERN1)</td>
<td>2892</td>
<td></td>
</tr>
<tr>
<td>package PATTERN2 PatternFunctions2</td>
<td>2894</td>
<td></td>
</tr>
<tr>
<td>PatternFunctions2 (PATTERN2)</td>
<td>2895</td>
<td></td>
</tr>
<tr>
<td>package PATMATCH PatternMatch</td>
<td>2896</td>
<td></td>
</tr>
<tr>
<td>PatternMatch (PATMATCH)</td>
<td>2897</td>
<td></td>
</tr>
<tr>
<td>package PMASS PatternMatchAssertions</td>
<td>2899</td>
<td></td>
</tr>
<tr>
<td>PatternMatchAssertions (PMASS)</td>
<td>2900</td>
<td></td>
</tr>
<tr>
<td>package PMFS PatternMatchFunctionSpace</td>
<td>2901</td>
<td></td>
</tr>
<tr>
<td>PatternMatchFunctionSpace (PMFS)</td>
<td>2902</td>
<td></td>
</tr>
<tr>
<td>package PMINS PatternMatchIntegerNumberSystem</td>
<td>2904</td>
<td></td>
</tr>
<tr>
<td>PatternMatchIntegerNumberSystem (PMINS)</td>
<td>2905</td>
<td></td>
</tr>
<tr>
<td>package INTPM PatternMatchIntegration</td>
<td>2907</td>
<td></td>
</tr>
<tr>
<td>PatternMatchIntegration (INTPM)</td>
<td>2908</td>
<td></td>
</tr>
<tr>
<td>package PMKERNEL PatternMatchKernel</td>
<td>2915</td>
<td></td>
</tr>
<tr>
<td>PatternMatchKernel (PMKERNEL)</td>
<td>2916</td>
<td></td>
</tr>
<tr>
<td>package PMLSAGG PatternMatchListAggregate</td>
<td>2918</td>
<td></td>
</tr>
<tr>
<td>PatternMatchListAggregate (PMLSAGG)</td>
<td>2919</td>
<td></td>
</tr>
<tr>
<td>package PMPLCAT PatternMatchPolynomialCategory</td>
<td>2921</td>
<td></td>
</tr>
<tr>
<td>PatternMatchPolynomialCategory (PMPLCAT)</td>
<td>2922</td>
<td></td>
</tr>
<tr>
<td>package PMDOWN PatternMatchPushDown</td>
<td>2924</td>
<td></td>
</tr>
<tr>
<td>PatternMatchPushDown (PMDOWN)</td>
<td>2925</td>
<td></td>
</tr>
<tr>
<td>package PMQFCAT PatternMatchQuotientFieldCategory</td>
<td>2927</td>
<td></td>
</tr>
<tr>
<td>PatternMatchQuotientFieldCategory (PMQFCAT)</td>
<td>2928</td>
<td></td>
</tr>
<tr>
<td>package PATRES2 PatternMatchResultFunctions2</td>
<td>2929</td>
<td></td>
</tr>
<tr>
<td>PatternMatchResultFunctions2 (PATRES2)</td>
<td>2930</td>
<td></td>
</tr>
<tr>
<td>package PMSYM PatternMatchSymbol</td>
<td>2931</td>
<td></td>
</tr>
<tr>
<td>PatternMatchSymbol (PMSYM)</td>
<td>2932</td>
<td></td>
</tr>
<tr>
<td>package PMTOOLS PatternMatchTools</td>
<td>2933</td>
<td></td>
</tr>
<tr>
<td>PatternMatchTools (PMTOOLS)</td>
<td>2934</td>
<td></td>
</tr>
<tr>
<td>package PERMAN Permanent</td>
<td>2938</td>
<td></td>
</tr>
<tr>
<td>Permanent (PERMAN)</td>
<td>2940</td>
<td></td>
</tr>
<tr>
<td>package PGE PermutationGroupExamples</td>
<td>2944</td>
<td></td>
</tr>
<tr>
<td>PermutationGroupExamples (PGE)</td>
<td>2945</td>
<td></td>
</tr>
<tr>
<td>package PICOERCE PiCoercions</td>
<td>2953</td>
<td></td>
</tr>
<tr>
<td>PiCoercions (PICOERCE)</td>
<td>2954</td>
<td></td>
</tr>
<tr>
<td>package PLOT1 PlotFunctions1</td>
<td>2955</td>
<td></td>
</tr>
<tr>
<td>Package Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>PlotFunctions1 (PLOT1)</td>
<td></td>
<td>2956</td>
</tr>
<tr>
<td>package PLOTTOOL PlotTools</td>
<td></td>
<td>2957</td>
</tr>
<tr>
<td>PlotTools (PLOTTOOL)</td>
<td></td>
<td>2958</td>
</tr>
<tr>
<td>package PRJALGPK ProjectiveAlgebraicSetPackage</td>
<td>ProjectiveAlgebraicSetPackage (PRJALGPK)</td>
<td>2960</td>
</tr>
<tr>
<td>package PTFUNC2 PointFunctions2</td>
<td>PointFunctions2 (PTFUNC2)</td>
<td>2965</td>
</tr>
<tr>
<td>package PTPACK PointPackage</td>
<td>PointPackage (PTPACK)</td>
<td>2966</td>
</tr>
<tr>
<td>package PFO PointsOfFiniteOrder</td>
<td>PointsOfFiniteOrder (PFO)</td>
<td>2967</td>
</tr>
<tr>
<td>package PFQOQ PointsOfFiniteOrderRational</td>
<td>PointsOfFiniteOrderRational (PFQOQ)</td>
<td>2968</td>
</tr>
<tr>
<td>package PFOTOOLS PointsOfFiniteOrderTools</td>
<td>PointsOfFiniteOrderTools (PFOTOOLS)</td>
<td>2969</td>
</tr>
<tr>
<td>package PLPKCRV PolynomialPackageForCurve</td>
<td>PolynomialPackageForCurve (PLPKCRV)</td>
<td>2970</td>
</tr>
<tr>
<td>package POLTOPO POLToPol</td>
<td>PolToPol (POLTOPO)</td>
<td>2971</td>
</tr>
<tr>
<td>package PGROEB PolyGroebner</td>
<td>PolyGroebner (PGROEB)</td>
<td>2972</td>
</tr>
<tr>
<td>package PAN2EXPR PolynomialAN2Expression</td>
<td>PolynomialAN2Expression (PAN2EXPR)</td>
<td>2973</td>
</tr>
<tr>
<td>package POLYLIFT PolynomialCategoryLifting</td>
<td>PolynomialCategoryLifting (POLYLIFT)</td>
<td>2974</td>
</tr>
<tr>
<td>package POLYCATQ PolynomialCategoryQuotientFunctions</td>
<td>PolynomialCategoryQuotientFunctions (POLYCATQ)</td>
<td>2975</td>
</tr>
<tr>
<td>package PCOMP PolynomialComposition</td>
<td>PolynomialComposition (PCOMP)</td>
<td>2976</td>
</tr>
<tr>
<td>package PDECOMP PolynomialDecomposition</td>
<td>PolynomialDecomposition (PDECOMP)</td>
<td>2977</td>
</tr>
<tr>
<td>package PFBR PolynomialFactorizationByRecursion</td>
<td>PolynomialFactorizationByRecursion (PFBR)</td>
<td>2978</td>
</tr>
<tr>
<td>package PFBRU PolynomialFactorizationByRecursionUnivariate</td>
<td>PolynomialFactorizationByRecursionUnivariate (PFBRU)</td>
<td>2979</td>
</tr>
<tr>
<td>package POLY2 PolynomialFunctions2</td>
<td>PolynomialFunctions2 (POLY2)</td>
<td>2980</td>
</tr>
<tr>
<td>package PGCD PolynomialGcdPackage</td>
<td>PolynomialGcdPackage (PGCD)</td>
<td>2981</td>
</tr>
<tr>
<td>package PINTERP PolynomialInterpolation</td>
<td>PolynomialInterpolation (PINTERP)</td>
<td>2982</td>
</tr>
<tr>
<td>package PINTERPA PolynomialInterpolationAlgorithms</td>
<td>PolynomialInterpolationAlgorithms (PINTERPA)</td>
<td>2983</td>
</tr>
<tr>
<td>package PNTHEORY PolynomialNumberTheoryFunctions</td>
<td>PolynomialNumberTheoryFunctions (PNTHEORY)</td>
<td>2984</td>
</tr>
<tr>
<td>package POLYROOT PolynomialRoots</td>
<td></td>
<td>2985</td>
</tr>
</tbody>
</table>
### CONTENTS

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolynomialRoots (POLYROOT)</td>
<td>3042</td>
</tr>
<tr>
<td>package PSETPK PolynomialSetUtilitiesPackage</td>
<td>3045</td>
</tr>
<tr>
<td>PolynomialSetUtilitiesPackage (PSETPK)</td>
<td>3046</td>
</tr>
<tr>
<td>package SOLVEFOR PolynomialSolveByFormulas</td>
<td>3063</td>
</tr>
<tr>
<td>PolynomialSolveByFormulas (SOLVEFOR)</td>
<td>3064</td>
</tr>
<tr>
<td>package PSQFR PolynomialSquareFree</td>
<td>3070</td>
</tr>
<tr>
<td>PolynomialSquareFree (PSQFR)</td>
<td>3071</td>
</tr>
<tr>
<td>package POLY2UP PolynomialToUnivariatePolynomial</td>
<td>3074</td>
</tr>
<tr>
<td>PolynomialToUnivariatePolynomial (POLY2UP)</td>
<td>3075</td>
</tr>
<tr>
<td>package LIMITPS PowerSeriesLimitPackage</td>
<td>3076</td>
</tr>
<tr>
<td>PowerSeriesLimitPackage (LIMITPS)</td>
<td>3077</td>
</tr>
<tr>
<td>package PREASSOC PrecomputedAssociatedEquations</td>
<td>3088</td>
</tr>
<tr>
<td>PrecomputedAssociatedEquations (PREASSOC)</td>
<td>3089</td>
</tr>
<tr>
<td>package PRIMARR2 PrimitiveArrayFunctions2</td>
<td>3091</td>
</tr>
<tr>
<td>PrimitiveArrayFunctions2 (PRIMARR2)</td>
<td>3092</td>
</tr>
<tr>
<td>package PRIMELT PrimitiveElement</td>
<td>3094</td>
</tr>
<tr>
<td>PrimitiveElement (PRIMELT)</td>
<td>3095</td>
</tr>
<tr>
<td>package ODEPRIM PrimitiveRatDE</td>
<td>3097</td>
</tr>
<tr>
<td>PrimitiveRatDE (ODEPRIM)</td>
<td>3098</td>
</tr>
<tr>
<td>package ODEPRRIC PrimitiveRatRicDE</td>
<td>3102</td>
</tr>
<tr>
<td>PrimitiveRatRicDE (ODEPRRIC)</td>
<td>3103</td>
</tr>
<tr>
<td>package PRINT PrintPackage</td>
<td>3109</td>
</tr>
<tr>
<td>PrintPackage (PRINT)</td>
<td>3110</td>
</tr>
<tr>
<td>package PSEUDLIN PseudoLinearNormalForm</td>
<td>3111</td>
</tr>
<tr>
<td>PseudoLinearNormalForm (PSEUDLIN)</td>
<td>3112</td>
</tr>
<tr>
<td>package PRS PseudoRemainderSequence</td>
<td>3115</td>
</tr>
<tr>
<td>PseudoRemainderSequence (PRS)</td>
<td>3116</td>
</tr>
<tr>
<td>package INTPAF PureAlgebraicIntegration</td>
<td>3136</td>
</tr>
<tr>
<td>PureAlgebraicIntegration (INTPAF)</td>
<td>3137</td>
</tr>
<tr>
<td>package ODEPAL PureAlgebraicLODE</td>
<td>3145</td>
</tr>
<tr>
<td>PureAlgebraicLODE (ODEPAL)</td>
<td>3146</td>
</tr>
<tr>
<td>package PUSHVAR PushVariables</td>
<td>3147</td>
</tr>
<tr>
<td>PushVariables (PUSHVAR)</td>
<td>3148</td>
</tr>
</tbody>
</table>

### 18 Chapter Q

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>package QALGSET2 QuasiAlgebraicSet2</td>
<td>3151</td>
</tr>
<tr>
<td>QuasiAlgebraicSet2 (QALGSET2)</td>
<td>3152</td>
</tr>
<tr>
<td>package QCMPACK QuasiComponentPackage</td>
<td>3155</td>
</tr>
<tr>
<td>QuasiComponentPackage (QCMPACK)</td>
<td>3156</td>
</tr>
<tr>
<td>package QFCAT2 QuotientFieldCategoryFunctions2</td>
<td>3164</td>
</tr>
<tr>
<td>QuotientFieldCategoryFunctions2 (QFCAT2)</td>
<td>3165</td>
</tr>
<tr>
<td>package QUATCT2 QuaternionCategoryFunctions2</td>
<td>3166</td>
</tr>
<tr>
<td>QuaternionCategoryFunctions2 (QUATCT2)</td>
<td>3168</td>
</tr>
</tbody>
</table>
19 Chapter R

package REP RadicalEigenPackage ........................................ 3171
    RadicalEigenPackage (REP) ........................................... 3172
package SOLVERAD RadicalSolvePackage .................................. 3176
    RadicalSolvePackage (SOLVERAD) ..................................... 3189
package RADUTIL RadixUtilities ............................................. 3196
    RadixUtilities (RADUTIL) ............................................. 3197
package RDIST RandomDistributions ........................................ 3198
    RandomDistributions (RDIST) .......................................... 3199
package RFDIST RandomFloatDistributions ................................ 3200
    RandomFloatDistributions (RFDIST) ................................... 3201
package RIDIST RandomIntegerDistributions ................................ 3203
    RandomIntegerDistributions (RIDIST) ................................. 3204
package RANDSRC RandomNumberSource .................................... 3206
    RandomNumberSource (RANDSRC) ..................................... 3207
package RATFACT RationalFactorize ....................................... 3208
    RationalFactorize (RATFACT) ......................................... 3209
package RF RationalFunction ................................................. 3211
    RationalFunction (RF) ................................................ 3212
package DEFINTRF RationalFunctionDefiniteIntegration ................. 3214
    RationalFunctionDefiniteIntegration (DEFINTRF) .................... 3215
package RFFACT RationalFunctionFactor ................................... 3217
    RationalFunctionFactor (RFFACT) ................................... 3218
package RFFACTOR RationalFunctionFactorizer ............................ 3219
    RationalFunctionFactorizer (RFFACTOR) .............................. 3220
package INTRF RationalFunctionIntegration ................................ 3222
    RationalFunctionIntegration (INTRF) ................................. 3223
package LIMITRF RationalFunctionLimitPackage .......................... 3224
    RationalFunctionLimitPackage (LIMITRF) ........................... 3225
package SIGNRF RationalFunctionSign ..................................... 3229
    RationalFunctionSign (SIGNRF) ...................................... 3230
package SUMRF RationalFunctionSum ...................................... 3232
    RationalFunctionSum (SUMRF) ........................................ 3238
package INTRAT RationalIntegration ...................................... 3240
    RationalIntegration (INTRAT) ....................................... 3241
package RINTERP RationalInterpolation .................................. 3243
    Introduction .......................................................... 3243
    Questions and Outlook ............................................... 3243
    RationalInterpolation (RINTERP) ..................................... 3244
package ODERAT RationalLODE .............................................. 3247
    RationalLODE (ODERAT) .............................................. 3248
package RATRET RationalRetractions ..................................... 3253
    RationalRetractions (RATRET) ...................................... 3254
package ODERTRIC RationalRicDE .......................................... 3255
    RationalRicDE (ODERTRIC) ........................................... 3256
package RURPK RationalUnivariateRepresentationPackage ............... 3262
RationalUnivariateRepresentationPackage (RURPK) ........................................ 3263
package POLUTIL RealPolynomialUtilitiesPackage ........................................... 3266
  RealPolynomialUtilitiesPackage (POLUTIL) .................................................. 3268
package REALSOLV RealSolvePackage .............................................................. 3270
  RealSolvePackage (REALSOLV) ......................................................................... 3274
package REAL0 RealZeroPackage ...................................................................... 3276
  RealZeroPackage (REAL0) ................................................................................ 3277
package REAL0Q RealZeroPackageQ ................................................................. 3283
  RealZeroPackageQ (REAL0Q) ........................................................................... 3284
package RMCAT2 RectangularMatrixCategoryFunctions2 ................................... 3286
  RectangularMatrixCategoryFunctions2 (RMCAT2) .................................................. 3287
package RECOP RecurrenceOperator .................................................................. 3288
  RecurrenceOperator (RECOP) ........................................................................... 3289
  Defining new operators ..................................................................................... 3291
  Recurrences ...................................................................................................... 3293
  Functional Equations ....................................................................................... 3297
package RDIV ReducedDivisor .......................................................................... 3300
  ReducedDivisor (RDIV) .................................................................................... 3301
package ODERED ReduceLODE ........................................................................... 3303
  ReduceLODE (ODERED) .................................................................................... 3304
package REDORDER ReductionOfOrder .................................................................. 3305
  ReductionOfOrder (REDORDER) ........................................................................ 3306
package RSDCMPK RegularSetDecompositionPackage ...................................... 3308
  RegularSetDecompositionPackage (RSDCMPK) .................................................. 3310
package RSETGCD RegularTriangularSetGcdPackage ....................................... 3316
  RegularTriangularSetGcdPackage (RSETGCD) .................................................... 3317
package REPDB RepeatedDoubling ...................................................................... 3324
  RepeatedDoubling (REPDB) ............................................................................. 3325
package REP2 RepeatedSquaring ....................................................................... 3328
  RepeatedSquaring (REPSQ) ............................................................................. 3328
package REP1 RepresentationPackage1 .................................................................. 3329
  RepresentationPackage1 (REP1) ...................................................................... 3330
package REP2 RepresentationPackage2 .................................................................. 3337
  RepresentationPackage2 (REP2) ...................................................................... 3338
package RESLATC ResolveLatticeCompletion .................................................. 3354
  ResolveLatticeCompletion (RESLATC) .............................................................. 3355
package RETSOL RetractSolvePackage .............................................................. 3356
  RetractSolvePackage (RETSOL) ........................................................................ 3357
package RFP RootsFindingPackage ..................................................................... 3359
  RootsFindingPackage (RFP) .............................................................................. 3360

20 Chapter S ................................................................................................. 3365
package SAERFFC SAERationalFunctionAlgFactor ............................................ 3365
  SAERationalFunctionAlgFactor (SAERFFC) ...................................................... 3366
package FORMULA1 ScriptFormulaFormat1 ....................................................... 3367
  ScriptFormulaFormat1 (FORMULA1) .............................................................. 3368
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGBIND2</td>
<td>SegmentBindingFunctions2</td>
<td>3369</td>
</tr>
<tr>
<td>SEG2</td>
<td>SegmentFunctions2 (SEG2)</td>
<td>3371</td>
</tr>
<tr>
<td>SAEFACT</td>
<td>SimpleAlgebraicExtensionAlgFactor (SAEFACT)</td>
<td>3373</td>
</tr>
<tr>
<td>SIMPAN</td>
<td>SimplifyAlgebraicNumberConvertPackage (SIMPAN)</td>
<td>3375</td>
</tr>
<tr>
<td>SMITH</td>
<td>SmithNormalForm (SMITH)</td>
<td>3377</td>
</tr>
<tr>
<td>SCACHE</td>
<td>SortedCache (SCACHE)</td>
<td>3383</td>
</tr>
<tr>
<td>SORTPAK</td>
<td>SortPackage (SORTPAK)</td>
<td>3386</td>
</tr>
<tr>
<td>SUP2</td>
<td>SparseUnivariatePolynomialFunctions2 (SUP2)</td>
<td>3388</td>
</tr>
<tr>
<td>SPECOUT</td>
<td>SpecialOutputPackage (SPECOUT)</td>
<td>3390</td>
</tr>
<tr>
<td>SFQCMKP</td>
<td>SquareFreeQuasiComponentPackage (SFQCMKP)</td>
<td>3393</td>
</tr>
<tr>
<td>SRDCMPK</td>
<td>SquareFreeRegularSetDecompositionPackage (SRDCMPK)</td>
<td>3403</td>
</tr>
<tr>
<td>SFRCGCD</td>
<td>SquareFreeRegularTriangularSetGcdPackage (SFRCGCD)</td>
<td>3410</td>
</tr>
<tr>
<td>MATSTOR</td>
<td>StorageEfficientMatrixOperations (MATSTOR)</td>
<td>3421</td>
</tr>
<tr>
<td>STREAM1</td>
<td>StreamFunctions1 (STREAM1)</td>
<td>3426</td>
</tr>
<tr>
<td>STREAM2</td>
<td>StreamFunctions2 (STREAM2)</td>
<td>3428</td>
</tr>
<tr>
<td>STREAM3</td>
<td>StreamFunctions3 (STREAM3)</td>
<td>3431</td>
</tr>
<tr>
<td>STINPROD</td>
<td>StreamInfiniteProduct (STINPROD)</td>
<td>3433</td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>StreamTaylorSeriesOperations (STTAYLOR)</td>
<td>3436</td>
</tr>
<tr>
<td>STNSR</td>
<td>StreamTensor (STNSR)</td>
<td>3446</td>
</tr>
<tr>
<td>STTF</td>
<td>StreamTranscendentalFunctions (STTF)</td>
<td>3448</td>
</tr>
<tr>
<td>STTFNC</td>
<td>StreamTranscendentalFunctionsNonCommutative (STTFNC)</td>
<td>3459</td>
</tr>
<tr>
<td>SCPKG</td>
<td>StructuralConstantsPackage (SCPKG)</td>
<td>3465</td>
</tr>
<tr>
<td>SHP</td>
<td>SturmHabichtPackage (SHP)</td>
<td>3469</td>
</tr>
<tr>
<td>Package</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>SUBRESP SubResultantPackage</td>
<td>3478</td>
<td></td>
</tr>
<tr>
<td>SupResultantPackage (SUBRESP)</td>
<td>3479</td>
<td></td>
</tr>
<tr>
<td>SUPFRACF SupFractionFactorizer</td>
<td>3482</td>
<td></td>
</tr>
<tr>
<td>SupFractionFactorizer (SUPFRACF)</td>
<td>3483</td>
<td></td>
</tr>
<tr>
<td>ODESYS SystemODESolver</td>
<td>3484</td>
<td></td>
</tr>
<tr>
<td>SystemODESolver (ODESYS)</td>
<td>3485</td>
<td></td>
</tr>
<tr>
<td>SYSSOLP SystemSolvePackage</td>
<td>3491</td>
<td></td>
</tr>
<tr>
<td>SystemSolvePackage (SYSSOLP)</td>
<td>3492</td>
<td></td>
</tr>
<tr>
<td>SGCF SymmetricGroupCombinatoricFunctions</td>
<td>3497</td>
<td></td>
</tr>
<tr>
<td>SymmetricGroupCombinatoricFunctions (SGCF)</td>
<td>3498</td>
<td></td>
</tr>
<tr>
<td>SYMFUNC SymmetricFunctions</td>
<td>3508</td>
<td></td>
</tr>
<tr>
<td>SymmetricFunctions (SYMFUNC)</td>
<td>3509</td>
<td></td>
</tr>
<tr>
<td>TABLBUMP TableauxBumpers</td>
<td>3511</td>
<td></td>
</tr>
<tr>
<td>TableauxBumpers (TABLBUMP)</td>
<td>3512</td>
<td></td>
</tr>
<tr>
<td>TBCMPPK TabulatedComputationPackage</td>
<td>3515</td>
<td></td>
</tr>
<tr>
<td>TabulatedComputationPackage (TBCMPPK)</td>
<td>3516</td>
<td></td>
</tr>
<tr>
<td>TANEXP TangentExpansions</td>
<td>3519</td>
<td></td>
</tr>
<tr>
<td>TangentExpansions (TANEXP)</td>
<td>3520</td>
<td></td>
</tr>
<tr>
<td>UTSSOL TaylorSolve</td>
<td>3521</td>
<td></td>
</tr>
<tr>
<td>TaylorSolve (UTSSOL)</td>
<td>3523</td>
<td></td>
</tr>
<tr>
<td>TEMUTL TemplateUtilities</td>
<td>3526</td>
<td></td>
</tr>
<tr>
<td>TemplateUtilities (TEMUTL)</td>
<td>3527</td>
<td></td>
</tr>
<tr>
<td>TEX1 TexFormat1</td>
<td>3528</td>
<td></td>
</tr>
<tr>
<td>TexFormat1 (TEX1)</td>
<td>3529</td>
<td></td>
</tr>
<tr>
<td>TOOLSIGN ToolsForSign</td>
<td>3530</td>
<td></td>
</tr>
<tr>
<td>ToolsForSign (TOOLSIGN)</td>
<td>3531</td>
<td></td>
</tr>
<tr>
<td>DRAW TopLevelDrawFunctions</td>
<td>3532</td>
<td></td>
</tr>
<tr>
<td>TopLevelDrawFunctions (DRAW)</td>
<td>3533</td>
<td></td>
</tr>
<tr>
<td>DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves</td>
<td>3540</td>
<td></td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForAlgebraicCurves (DRAWCURV)</td>
<td>3541</td>
<td></td>
</tr>
<tr>
<td>DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions</td>
<td>3544</td>
<td></td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForCompiledFunctions (DRAWCFUN)</td>
<td>3545</td>
<td></td>
</tr>
<tr>
<td>DRAWPT TopLevelDrawFunctionsForPoints</td>
<td>3558</td>
<td></td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForPoints (DRAWPT)</td>
<td>3559</td>
<td></td>
</tr>
<tr>
<td>TOPSP TopLevelThreeSpace</td>
<td>3561</td>
<td></td>
</tr>
<tr>
<td>TopLevelThreeSpace (TOPSP)</td>
<td>3562</td>
<td></td>
</tr>
<tr>
<td>INTHETR TranscendentalHermiteIntegration</td>
<td>3563</td>
<td></td>
</tr>
<tr>
<td>TranscendentalHermiteIntegration (INTHETR)</td>
<td>3564</td>
<td></td>
</tr>
<tr>
<td>INTRR TranscendentalIntegration</td>
<td>3566</td>
<td></td>
</tr>
<tr>
<td>TranscendentalIntegration</td>
<td>3567</td>
<td></td>
</tr>
<tr>
<td>TRMANIP TranscendentalManipulations</td>
<td>3576</td>
<td></td>
</tr>
<tr>
<td>TranscendentalManipulations (TRMANIP)</td>
<td>3577</td>
<td></td>
</tr>
<tr>
<td>ltrigs function</td>
<td>3585</td>
<td></td>
</tr>
<tr>
<td>RDETR TranscendentalRischDE</td>
<td>3591</td>
<td></td>
</tr>
</tbody>
</table>
TranscendentalRischDE (RDETR) .......................................................... 3592
package RDETRS TranscendentalRischDESystem .............................. 3596
  TranscendentalRischDESystem (RDETRS) ....................................... 3597
package SOLVETRA TransSolvePackage ........................................... 3602
  TransSolvePackage (SOLVETRA) ............................................... 3608
package SOLVESER TransSolvePackageService ................................ 3619
  TransSolvePackageService (SOLVESER) ...................................... 3620
package TRIMAT TriangularMatrixOperations .................................. 3623
  TriangularMatrixOperations (TRIMAT) ....................................... 3624
package TRIGMNIP TrigonometricManipulations ............................... 3626
  TrigonometricManipulations (TRIGMNIP) .................................... 3627
package TUBETOOL TubePlotTools ................................................. 3630
  TubePlotTools (TUBETOOL) ................................................... 3631
package CLIP TwoDimensionalPlotClipping .................................. 3634
  TwoDimensionalPlotClipping (CLIP) ...................................... 3635
package TWOFACT TwoFactorize ................................................. 3641
  TwoFactorize (TWOFACT) .................................................... 3642
package UNIFACT UnivariateFactorize ........................................ 3649
  UnivariateFactorize (UNIFACT) ............................................ 3650
package UFPS1 UnivariateFormalPowerSeriesFunctions ...................... 3657
  UnivariateFormalPowerSeriesFunctions (UFPS1) .......................... 3658
package ULS2 UnivariateLaurentSeriesFunctions2 ........................... 3659
  UnivariateLaurentSeriesFunctions2 (ULS2) ................................ 3660
package UPOLYC2 UnivariatePolynomialCategoryFunctions2 ................. 3661
  UnivariatePolynomialCategoryFunctions2 (UPOLYC2) ...................... 3662
package UPCDEN UnivariatePolynomialCommonDenominator .................... 3663
  UnivariatePolynomialCommonDenominator (UPCDEN) ....................... 3664
package UPDECOMP UnivariatePolynomialDecompositionPackage ............. 3665
  UnivariatePolynomialDecompositionPackage (UPDECOMP) .................. 3666
package UPDIVP UnivariatePolynomialDivisionPackage ...................... 3669
  UnivariatePolynomialDivisionPackage (UPDIVP) .......................... 3670
package UP2 UnivariatePolynomialFunctions2 ................................ 3671
  UnivariatePolynomialFunctions2 (UP2) .................................... 3672
package UPMP UnivariatePolynomialMultiplicationPackage .................. 3673
  UnivariatePolynomialMultiplicationPackage (UPMP) ....................... 3674
package UPSQFREE UnivariatePolynomialSquareFree ......................... 3677
  UnivariatePolynomialSquareFree (UPSQFREE) ............................... 3678
package UPXS2 UnivariatePuiseuxSeriesFunctions2 .......................... 3680
  UnivariatePuiseuxSeriesFunctions2 (UPXS2) ................................ 3681
package OREPCTO UnivariateSkewPolynomialCategoryOps ..................... 3683
  UnivariateSkewPolynomialCategoryOps (OREPCTO) .......................... 3684
package UTS2 UnivariateTaylorSeriesFunctions2 ............................ 3687
  UnivariateTaylorSeriesFunctions2 (UTS2) ................................ 3688
package UTSODE UnivariateTaylorSeriesODESolver ........................... 3689
## CONTENTS

<table>
<thead>
<tr>
<th>Package/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnivariateTaylorSeriesODESolver (UTSODE)</td>
<td>3690</td>
</tr>
<tr>
<td>package UNISEG2 UniversalSegmentFunctions2</td>
<td>3693</td>
</tr>
<tr>
<td>UniversalSegmentFunctions2 (UNISEG2)</td>
<td>3694</td>
</tr>
<tr>
<td>package UDPO UserDefinedPartialOrdering</td>
<td>3695</td>
</tr>
<tr>
<td>UserDefinedPartialOrdering (UDPO)</td>
<td>3696</td>
</tr>
<tr>
<td>package UDVO UserDefinedVariableOrdering</td>
<td>3698</td>
</tr>
<tr>
<td>UserDefinedVariableOrdering (UDVO)</td>
<td>3699</td>
</tr>
<tr>
<td>package UTSODETL UTSodetools</td>
<td>3700</td>
</tr>
<tr>
<td>UTSodetools (UTSODETL)</td>
<td>3701</td>
</tr>
<tr>
<td>package POLYVEC U32VectorPolynomialOperations</td>
<td>3703</td>
</tr>
<tr>
<td>U32VectorPolynomialOperations (POLYVEC)</td>
<td>3704</td>
</tr>
<tr>
<td>package VECTOR2 VectorFunctions2</td>
<td>3717</td>
</tr>
<tr>
<td>VectorFunctions2 (VECTOR2)</td>
<td>3718</td>
</tr>
<tr>
<td>package VIEWDEF ViewDefaultsPackage</td>
<td>3719</td>
</tr>
<tr>
<td>ViewDefaultsPackage (VIEWDEF)</td>
<td>3720</td>
</tr>
<tr>
<td>package VIEW ViewportPackage</td>
<td>3725</td>
</tr>
<tr>
<td>ViewportPackage (VIEW)</td>
<td>3726</td>
</tr>
<tr>
<td>package WEIER WeierstrassPreparation</td>
<td>3731</td>
</tr>
<tr>
<td>WeierstrassPreparation (WEIER)</td>
<td>3732</td>
</tr>
<tr>
<td>package WFFINTBS WildFunctionFieldIntegralBasis</td>
<td>3735</td>
</tr>
<tr>
<td>WildFunctionFieldIntegralBasis (WFFINTBS)</td>
<td>3736</td>
</tr>
<tr>
<td>package XEXPPKG XExponentialPackage</td>
<td>3743</td>
</tr>
<tr>
<td>XExponentialPackage (XEXPPKG)</td>
<td>3744</td>
</tr>
<tr>
<td>package ZDSOLVE ZeroDimensionalSolvePackage</td>
<td>3749</td>
</tr>
<tr>
<td>ZeroDimensionalSolvePackage (ZDSOLVE)</td>
<td>3812</td>
</tr>
<tr>
<td>28 Chunk collections</td>
<td>3823</td>
</tr>
<tr>
<td>29 Index</td>
<td>3835</td>
</tr>
</tbody>
</table>
New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation’s website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we’ve broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We’ve also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I’m looking forward to future milestones.

With that in mind I’ve introduced the theme of the “30 year horizon”. We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How will we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The “30 year horizon” is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))
Chapter 1

Chapter Overview

This book contains the packages in Axiom, in alphabetical order.

Each package has an associated ‘dotpic’ chunk which only lists the packages, categories, and packages that are in the layer immediately below in the build order. For the full list see the algebra Makefile where this information is maintained.

Each package is preceded by a picture. The picture indicates several things. The colors indicate whether the name refers to a category, package, or package. An ellipse means that the name refers to something in the bootstrap set. Thus,

```
Category  Domain  Package  CABMON
```
Chapter 2

Chapter A

package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis

--- AffineAlgebraicSetComputeWithGroebnerBasis.input ---

)set break resume
)sys rm -f AffineAlgebraicSetComputeWithGroebnerBasis.output
)spool AffineAlgebraicSetComputeWithGroebnerBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineAlgebraicSetComputeWithGroebnerBasis

--R AffineAlgebraicSetComputeWithGroebnerBasis(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: DirectProductCategory(#(symb),NonNegativeInteger),ProjPt: ProjectiveSpaceCategory(K)) is a package constructor

--R Abbreviation for AffineAlgebraicSetComputeWithGroebnerBasis is AFALGGRO

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGGRO

--R------------------------------------------ Operations ------------------------------------------

--R affineAlgSet : List(PolyRing) -> Union(List(ProjPt),"failed",Infinite,Integer)

--R affineRationalPoints : (PolyRing,PositiveInteger) -> List(ProjPt)

--R affineSingularPoints : PolyRing -> Union(List(ProjPt),"failed",Infinite,Integer)

--R

--E 1

)spool
)lisp (bye)
AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO)

| AFALGGRO |
|———|

| DIRPCAT | PFECAT |
|——— | ——
| Exports: | affineAlgSet  affineRationalPoints  affineSingularPoints |

— package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis —

)abbrev package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt):Exports_
== Implementation where
K  : Field
symb: List(Symbol)
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)

OF ==> OutputForm
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
RFP ==> RootsFindingPackage
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)

Exports ==> with

  affineAlgSet: List PolyRing -> _
  Union(List(ProjPt),"failed","Infinite",Integer)

  affineSingularPoints : PolyRing -> _
  Union(List(ProjPt),"failed","Infinite",Integer)

  affineRationalPoints: (PolyRing,PI) -> List ProjPt
  ++ \axiom{rationalPoints(f,d)} returns all points on the curve
  ++ \axiom{f} in the extension of the ground field of degree \axiom{d}.
  ++ For \axiom{d > 1} this only works if \axiom{K} is a
  ++ \axiomType{LocallyAlgebraicallyClosedField}

Implementation ==> add

ss2:List Symbol:= [X1,X2]

DD ==> DistributedMultivariatePolynomial(ss2,K)
LexE ==> DirectProduct(#ss2,NonNegativeInteger)
OV2 ==> OrderedVariableList(ss2)
InGB ==> InterfaceGroebnerPackage(K,ss2,LexE,OV2,DD)

affineAlgSetLocal : List DD -> _
  Union(List(ProjPt),"failed","Infinite",Integer)

import PPFC1
import PolyRing
import ProjPt

listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]

polyToYX1 : PolyRing -> DD
-- NOTE : polyToYX1 set the last variable to 1 and swap the 1st and 2nd var
-- so that a call to grobner will eliminate the second var before the
-- first one
-- 23/10/98 : Ce n'est plus vrai. La fonction a ete "repare".
-- A priori ce la ne creait pas de bug, car on tenait compte de
-- cette particularite dans la fonction affineAlgSetLocal.
-- cette derniere fct a aussi ete "ajuste"
-- 27/10/98
-- Ce n'est pas vrai !! Il faut trouve X d'abord et ensuite Y !!
-- sinon tout sr la notion de places distinguee fout le camp !!!

polyToX10 : PolyRing -> SUP(K)

--fuctions de resolution de sys. alg. de dim 0

if K has FiniteFieldCategory then

  affineRationalPoints(crv:PolyRing,extdegree:PI):List(ProjPt) ==
  --The code of this is almost the same as for algebraicSet
  --We could just construct the ideal and call algebraicSet
  --Should we do that? This might be a bit faster.

  listPtsIdl:List(ProjPt):= empty()
  
  x:= monomial(1,directProduct(vector([1,0])$Vector(NNI)))$DD
  y:= monomial(1,directProduct(vector([0,1])$Vector(NNI)))$DD

  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)$K
    q:= size()$K
    px:= x**(q**extdegree) - x
    py:= y**(q**extdegree) - y

  crvXY1 := polyToYX1 crv
  rpts:= affineAlgSetLocal([crvXY1,px,py])

  -- si les 3 tests qui suivent ne sont pas la,
  -- alors ca ne compile pas !!!! ???
  rpts case "failed" =>
    error "failed: From affineRationalPoints in AFALGGRO,"
    rpts case "Infinite" =>
    error "Infinite: From affineRationalPoints in AFALGGRO,"
  rpts case Integer =>
    error "Integer: From affineRationalPoints in AFALGGRO,"
  rpts case List(ProjPt) => rpts
  error "Unknown: From affineRationalPoints in AFALGGRO,"

  affineSingularPoints(crb)==
  F:= polyToYX1 crb
  Fx:=differentiate(F,index(1)$OV2)
  Fy:=differentiate(F,index(2)$OV2)
  affineAlgSetLocal([F,Fx,Fy])
affineAlgSet(ideal : List PolyRing )==
    idealXY1 := [polyToYX1 pol for pol in ideal]
    affineAlgSetLocal idealXY1

-- fonctions de resolution de sys. alg. de dim 0
affineAlgSetLocal(idealToXY1:List DD ) ==
    listPtsIdl:=List(ProjPt)
    idealGroXY1:=groebner(idealToXY1)$InGB
    listZeroY:List(K):=empty()
    listZeroX:List(K):=empty()
    listOfExtDeg:List(Integer):=empty()
    polyZeroX:DD:=last(idealGroXY1)
    member?(index(1)$OV2, variables(polyZeroX)$DD) =>
        print(('"The number of point in the algebraic set is not finite")::OF)
        print(('"or the curve is not absolutely irreducible.")::OF)
        error "Have a nice day"
    -- now we find all of the projective points where z ^= 0
    recOfZerosX:=distinguishedRootsOf(univariate(polyZeroX),1$K)$RFP(K)
    -- HERE CHANGE
    degExtX:=recOfZerosX.extDegree
    listZeroX:List K := recOfZerosX.zeros
    listOfExtDeg:=cons(degExtX,listOfExtDeg)
    for a in listZeroX repeat
        tjeker := [(eval(f,index(2)$OV2,a)$DD) for f in idealGroXY1]
        idealGroaXb1 := [univariate(f)$DD for f in tjeker]
        recOfZerosOfIdeal:=distinguishedCommonRootsOf(idealGroaXb1,a)$RFP(K)
        listZeroY:= recOfZerosOfIdeal.zeros
        listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree,listOfExtDeg)
        listPtsIdl:=
            concat( [projectivePoint([a,b,1]) for b in listZeroY] ,listPtsIdl)
    degExt:=lcm listOfExtDeg
    zero?(degExt) =>
        print(('"------- Infinite number of points -------")::OF)
        "Infinite"
    ^one?(degExt) =>
        print(('"You need an extension of degree")::OF)
        print(degExt::OF)
        degExt
        listPtsIdl

polyToYX1(pol)==
    zero?(pol) => 0
    dd:= degree pol
    lc:= leadingCoefficient pol
    pp:= parts dd
    ppr:= rest reverse pp
    ppv:Vector(NNI):= vector ppr
    eppr:=directProduct(ppv)$LexE
    monomial(lc,eppr)$DD + polyToYX1 reductum pol
CHAPTE R 2.  CHAPTE R A

polyToX10(pol) =
  zero?(pol) \Rightarrow 0
  dd := degree pol
  lc := leadingCoefficient pol
  pp := parts dd
  lp := last pp
  ~zero?(lp) \Rightarrow polyToX10 reductum pol
  e1 := pp.1
  monomial(lc, e1)$SUP(K) + polyToX10 reductum pol

---

AFALGGRO.dotabb

"AFALGGRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGGRO"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"AFALGGRO" -> "DIRPCAT"
"AFALGGRO" -> "PFECAT"

---

package AFALGRES AffineAlgebraicSetComputeWithResultant

--- AffineAlgebraicSetComputeWithResultant.input ---

)set break resume
)sys rm -f AffineAlgebraicSetComputeWithResultant.output
)spool AffineAlgebraicSetComputeWithResultant.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineAlgebraicSetComputeWithResultant
  --R
  --R AffineAlgebraicSetComputeWithResultant(K: Field, symb: List(Symbol), PolyRing: PolynomialCategory K, ...) is a constructor
  --R Abbreviation for AffineAlgebraicSetComputeWithResultant is AFALGRES
  --R This constructor is exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGRES
--R
--R-------------------------------- Operations --------------------------------
AffineAlgebraicSetComputeWithResultant (AFALGRES)

Exports:

- affineAlgSet
- affineAlgSetLocal
- affineRationalPoints
- affineSingularPoints
- allPairsAmong
- polyRing2UPUP
| package AFALGRES AffineAlgebraicSetComputeWithResultant |

)abbrev package AFALGRES AffineAlgebraicSetComputeWithResultant
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt):Ex==Impl where
K : Field
symb: List(Symbol)
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)

SUP ==> SparseUnivariatePolynomial
UPUP ==> SUP(SUP(K))
NNI ==> NonNegativeInteger
RFP ==> RootsFindingPackage

Ex ==> with

  affineSingularPoints: PolyRing -> _
      Union(List(ProjPt),"failed","Infinite",Integer)

  affineSingularPoints: UPUP -> _
      Union(List(ProjPt),"failed","Infinite",Integer)

  affineAlgSetLocal: List UPUP -> _
      Union(List(ProjPt),"failed","Infinite",Integer)

  affineAlgSet: List PolyRing -> _
      Union(List ProjPt ,"failed","Infinite",Integer)

  polyRing2UPUP: PolyRing -> UPUP

  allPairsAmong: List UPUP -> List List UPUP

  affineRationalPoints: (PolyRing, PositiveInteger) -> _
      Union(List(ProjPt),"failed","Infinite",Integer)

Impl ==> add

import ProjPt

evAtcoef: (UPUP,K) -> SUP(K)
evAtcoef(pol,a)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  monomial( lc(a), dd )$SUP(K) + evAtcoef( reductum(pol), a )

polyRing2UPUP(pol)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  pp:= parts dd
  monomial(monomial(lc,pp.1)$SUP(K),pp.2)$UPUP+polyRing2UPUP(reductum(pol))

if K has FiniteFieldCategory then
  affineRationalPoints(crv:PolyRing,extdegree:PositiveInteger) ==
  listPtsIdl:List(ProjPt):= empty()
  x:= monomial(1,directProduct(vector([1,0,0])$Vector(NNI)))$PolyRing
  y:= monomial(1,directProduct(vector([0,1,0])$Vector(NNI)))$PolyRing
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)$K
  q:= size()$K
  px:= x**(q**extdegree) - x
  py:= y**(q**extdegree) - y
  rpts:= affineAlgSet([crv,px,py])
  -- si les 3 tests qui suivent ne sont pas la,
  -- alors ca ne compile pas !!! ???
  rpts case "failed" => _
  error "case failed: From affineRationalPoints in AFALGRES"
  rpts case "Infinite" => _
  error "case infinite: From affineRationalPoints in AFALGRES"
  rpts case Integer => _
  error "case Integer: From affineRationalPoints in AFALGRES"
  rpts case List(ProjPt) => rpts
  error "case unknown: From affineRationalPoints in AFALGRES"

allPairsAmong(lp)==
  #lp = 2 => [lp]
  rlp:=rest lp
  subL:= allPairsAmong rlp
  pol:=first lp
  frontL:= [[pol,p] for p in rlp]
  concat( frontL , subL )

affineSingularPoints(pol:PolyRing)==
  affineSingularPoints( polyRing2UPUP pol )

affineSingularPoints(pol:UPUP)==
  ground? pol => "failed"
lc := coefficients pol
lcb := [ ground?( c )$SUP(K) for c in lc ]
reduce("and", lcb) => "failed"
dy:=differentiate(pol)
dx:=map(differentiate$SUP(K),pol)
affineAlgSetLocal( [ pol, dy, dx ] )

resultantL: List UPUP -> SUP(K)
resultantL(lp)==
g:=first lp
h:= last lp
resultant(g,h)

affineAlgSet(lpol:List PolyRing)==
affineAlgSetLocal( [ polyRing2UPUP pol for pol in lpol ] )

affineAlgSetLocal(lpol:List UPUP)==
listPtsIdl:List(ProjPt)
allP:= allPairsAmong lpol
beforGcd:List SUP(K) := [resultantL(lp) for lp in allP]
polyZeroX:SUP(K):=gcd beforGcd
zero? polyZeroX => "failed"
listZeroY:List(K):=empty()
listZeroX:List(K):=empty()
recOfZerosX:=distinguishedRootsOf(polyZeroX,1$K)$RFP(K)
degExtX:=recOfZerosX.extDegree
listZeroX:List K := recOfZerosX.zeros
listOfExtDeg:List(Integer):=empty()
listOfExtDeg:=cons(degExtX,listOfExtDeg)
lpolEval:List SUP(K)
for a in listZeroX repeat
  lpolEval := [ evAtcoef(p,a) for p in lpol ]
recOfZerosOfIdeal:=distinguishedCommonRootsOf( lpolEval ,a)$RFP(K)
listZeroY:= recOfZerosOfIdeal.zeros
listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree,listOfExtDeg)
listPtsIdl:=
  concat( [projectivePoint([a,b,1]) for b in listZeroY] ,listPtsIdl)
degExt:=lcm listOfExtDeg
zero?(degExt) =>
  print("AFALGRES:Infinite number of points")::OutputForm
  "Infinite"
"one?(degExt) =>
  print("AFALGRES:You need an extension of degree")::OutputForm
  print(degExt::OutputForm)
degExt
listPtsIdl
--- AfAlgRes.dotabb ---

"AFALGRES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGRES"]
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"AFALGRES" -> "RFP"

---

package AF AlgebraicFunction

--- AlgebraicFunction.input ---

)set break resume
)sys rm -f AlgebraicFunction.output
)spool AlgebraicFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicFunction
--E 1

)spool
)lisp (bye)

---

--- AlgebraicFunction.help ---

====================================================================
AlgebraicFunction examples
====================================================================

This package provides algebraic functions over an integral domain.

See Also:
o )show AlgebraicFunction

---
AlgebraicFunction (AF)

Exports:
belong? droot definingPolynomial inrootof iroot
minPoly operator rootOf ?**?

— package AF AlgebraicFunction —

)abbrev package AF AlgebraicFunction
++ Author: Manuel Bronstein
++ Date Created: 21 March 1988
++ Date Last Updated: 11 November 1993
++ Description:
++ This package provides algebraic functions over an integral domain.

AlgebraicFunction(R, F): Exports == Implementation where
R: Join(OrderedSet, IntegralDomain)
F: FunctionSpace R

SE ==> Symbol
Z ==> Integer
Q ==> Fraction Z
OP ==> BasicOperator
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
UPR ==> SparseUnivariatePolynomial R
ALGOP ==> "%alg"
SPECIALDISP ==> "%specialDisp"
SPECIALDIFF ==> "%specialDiff"

Exports == with
rootOf : (UP, SE) -> F
++ rootOf(p, y) returns y such that \( p(y) = 0 \).
++ The object returned displays as \( 'y \).
operator: OP -> OP
++ operator(op) returns a copy of \spad{op} with the domain-dependent
++ properties appropriate for \texttt{F}.
++ Error: if \texttt{op} is not an algebraic operator, that is,
++ an nth root or implicit algebraic operator.

\begin{verbatim}
belong?: OP -> Boolean
++ belong?(op) is true if \texttt{op} is an algebraic operator, that is,
++ an nth root or implicit algebraic operator.
inrootof: (UP, F) -> F
++ inrootof(p, x) should be a non-exported function.
-- un-export when the compiler accepts conditional local functions!
droot : List F -> OutputForm
++ droot(l) should be a non-exported function.
-- un-export when the compiler accepts conditional local functions!

if R has RetractableTo Integer then
"**" : (F, Q) -> F
++ x ** q is \texttt{x} raised to the rational power \texttt{q}.

minPoly: K -> UP
++ minPoly(k) returns the defining polynomial of \texttt{k}.

definingPolynomial: F -> F
++ definingPolynomial(f) returns the defining polynomial of \texttt{f}
++ as an element of \texttt{F}.
++ Error: if \texttt{f} is not a kernel.
iroot : (R, Z) -> F
++ iroot(p, n) should be a non-exported function.
-- un-export when the compiler accepts conditional local functions!
\end{verbatim}

Implementation ==> add

\begin{verbatim}
ialg : List F -> F
ivalg: (List F, SE) -> F
dalg : List F -> OutputForm

opalg := operator("rootOf":::Symbol)$CommonOperators
oproot := operator("nthRoot":::Symbol)$CommonOperators

belong? op == has?(op, ALGOP)
dalg l == second(l)::OutputForm

rootOf(p, x) ==
k := kernel(x)$K
(r := retractIfCan(p)@Union(F, "failed")) case "failed" =>
inrootof(p, k::F)
n := numer(f := univariate(r::F, k))
degree denom f > 0 => error "rootOf: variable appears in denom"
inrootof(n, k::F)

ivalg(l, x) ==
p := numer univariate(first l, retract(second l)$K)
alpha := kernel(opalg, l)
- (map((s:F):F +-> differentiate(s, x), p) alpha)_
  / ((differentiate p) alpha)
\end{verbatim}
ialg l ==
f := univariate(p := first l, retract(x := second l)@K)
derg d denom f > 0 => error "roofOf: variable appears in denom"
inrootof(numer f, x)

operator op ==
is?(op, "rootOf"::Symbol) => opalg
is?(op, "nthRoot"::Symbol) => oproot
error "Unknown operator"

if R has AlgebraicallyClosedField then
UP2R: UP -> Union(UPR, "failed")
inrootof(q, x) ==
monomial? q => 0
(d := degree q) <= 0 => error "rootOf: constant polynomial"
-- one? d => - leadingCoefficient(reductum q) / leadingCoefficient q
(d = 1) => - leadingCoefficient(reductum q) / leadingCoefficient q
((rx := retractIfCan(x)@Union(SE, "failed")) case SE) and
((r := UP2R q) case UPR) => rootOf(r::UPR, rx::SE)::F
kernel(opalg, [q x, x])

UP2R p ==
ans:UPR := 0
while p ^= 0 repeat
(r := retractIfCan(leadingCoefficient p)@Union(R, "failed"))
case "failed" => return "failed"
ans := ans + monomial(r::R, degree p)
p := reductum p
ans

else
inrootof(q, x) ==
monomial? q => 0
(d := degree q) <= 0 => error "rootOf: constant polynomial"
-- one? d => - leadingCoefficient(reductum q) / leadingCoefficient q
(d = 1) => - leadingCoefficient(reductum q) / leadingCoefficient q
kernel(opalg, [q x, x])

evaluate(opalg, ialg)$BasicOperatorFunctions1(F)
setProperty(opalg, SPECIALDIFF, dvalg@((List F, SE) -> F) pretend None)
setProperty(opalg, SPECIALDISP, dalg@((List F) -> OutputForm) pretend None)

if R has RetractableTo Integer then
import PolynomialRoots(IndexedExponents K, K, R, P, F)
dumvar := "%%var"::Symbol::F
PACKAGE AF ALGEBRAICFUNCTION

17

lzero : List F -> F
dvroot : List F -> F
inroot : List F -> F
hackroot: (F, Z) -> F
inroot0 : (F, Z, Boolean, Boolean) -> F

lzero l == 0
droot l ==
x := first(l)::OutputForm
(n := retract(second l)@Z) = 2 => root x
root(x, n::OutputForm)
dvroot l ==
n := retract(second l)@Z
(first(l) ** ((1 - n) / n)) / (n::F)
x ** q ==
qr := divide(numer q, denom q)
x ** qr.quotient * inroot([x, (denom q)::F]) ** qr.remainder

hackroot(x, n) ==
(n = 1) or (x = 1) => x
(((dx := denom x) ^= 1) and
 (positive?(rx))
 => hackroot((numer x)::F, n)/hackroot(rx::Integer::F, n)
(x = -1) and n = 4 =>
((-1::F) ** (1::Q / 2::Q) + 1) / ((2::F) ** (1::Q / 2::Q))
kernel(oproot, [x, n::F])
inroot l ==
zero?(n := retract(second l)@Z) => error "root: exponent = 0"
-- one?(x := first l) or one? n => x
-- (x := first l) = 1 or (n = 1) => x
(r := retractIfCan(x)@Union(R, ["failed"])) case R => iroot(r::R, n)
(u := isExpt(x, oproot)) case Record(var:K, exponent:Z) =>
(first argument(pr.var)) **
 (pr.exponent /$Fraction(Z)
  (n * retract(second argument(pr.var))@Z))
inroot0(x, n, false, false)
inroot0(x, n, num?, den?) ==
nr:Union(Z, ["failed"]) := (num? => "failed"; retractIfCan numerator x)
rn:Union(Z, ["failed"]) := (den? => "failed"; retractIfCan denominator x)
(rn case Z) and (rd case Z) =>
rec := qroot(rn::Z / rd::Z, n::NonNegativeInteger)
rec.coef * hackroot(rec.radicand, rec.exponent)

rn case Z =>
rec := qroot(rn::Z::Fraction(Z), n::NonNegativeInteger)
rec.coef * inroot0((rec.radicand**(n exquo rec.exponent)::Z)
/ (denom(x)::F), n, true, den?)

rd case Z =>
rec := qroot(rd::Z::Fraction(Z), n::NonNegativeInteger)
inroot0((numer(x)::F) /
(rec.radicand ** (n exquo rec.exponent)::Z),
n, num?, true) / rec.coef

hackroot(x, n)

if R has AlgebraicallyClosedField then iroot(r, n) == nthRoot(r, n)::F
else
iroot0: (R, Z) -> F

if R has RadicalCategory then
if R has imaginary(): -> R then iroot(r, n) == nthRoot(r, n)::F
else
iroot(r, n) ==
odd? n or r >= 0 => nthRoot(r, n)::F
iroot0(r, n)

else iroot(r, n) == iroot0(r, n)

iroot0(r, n) ==
rec := rroot(r, n::NonNegativeInteger)
rec.coef * hackroot(rec.radicand, rec.exponent)

definingPolynomial x ==
(r := retractIfCan(x)@Union(K, "failed")) case K =>
is?(k := r::K, opalg) => first argument k
is?(k, oproot) =>
dumvar ** retract(second argument k)@Z - first argument k
dumvar - x
dumvar - x

minPoly k ==
is?(k, opalg) =>
numer univariate(first argument k,
retract(second argument k)@K)
is?(k, oproot) =>
monomial(1, retract(second argument k)@Z :: NonNegativeInteger)
- first(argument k)::UP
monomial(1, 1) - k::F::UP

evaluate(oproot, inroot)$BasicOperatorFunctions1(F)
derivative(oproot, [dvroot, lzero])
else -- R is not retractable to Integer
droot l ==
x := first(l)::OutputForm
(n := second l) = 2::F => root x
root(x, n::OutputForm)

minPoly k ==
  is?(k, opalg) =>
    numer univariate(first argument k, 
    retract(second argument k)@K)
  monomial(1, 1) - k::F::UP

setProperty(oproot, SPECIALDISP,
  droot@List F -> OutputForm pretend None)

— AF.dotabb —

"AF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"AF" -> "FS"
"AF" -> "ACF"

— package INTHERAL AlgebraicHermiteIntegration —

AlgebraicHermiteIntegration (INTHERAL)

— AlgebraicHermiteIntegration.input —

)set break resume
)sys rm -f AlgebraicHermiteIntegration.output
)spool AlgebraicHermiteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicHermiteIntegration
--E 1
Algebraic Hermite reduction.

See Also:
- `)show AlgebraicHermiteIntegration`

Exports:
HermiteIntegrate

---

AlgebraicHermiteIntegration(F, UP, UPUP, R):Exports == Implementation where
F : Field
UP : UnivariatePolynomialCategory F
UPUP: UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F, UP, UPUP)
N ==> NonNegativeInteger
RF ==> Fraction UP

Exports ==> with
    HermiteIntegrate: (R, UP -> UP) -> Record(answer:R, logpart:R)
++ HermiteIntegrate(f, ') returns \spad{[g,h]} such that
++ \spad{f = g' + h} and h has a only simple finite normal poles.

Implementation ==> add
    localsolve: (Matrix UP, Vector UP, UP) -> Vector UP

-- the denominator of f should have no prime factor P s.t. P | P'
-- (which happens only for P = t in the exponential case)
HermiteIntegrate(f, derivation) ==
    ratsform := 0
    n := rank()
    m := transpose((mat:= integralDerivationMatrix derivation).num)
    inum := (cform := integralCoordinates f).num
    iden := (iden := cform.den) exquo (e := mat.den) case "failed" then
        iden := (coef := (e exquo gcd(e, iden))::UP) * iden
        inum := coef * inum
    for trm in factors squareFree iden | (j:= trm.exponent) > 1 repeat
        u' := (u := (iden exquo (v := trm.factor)**(j::N))::UP) * derivation v
        sys := ((u * v) exquo e)::UP * m
        nn := minRowIndex sys - minIndex inum
        while j > 1 repeat
            j := j - 1
            p := - j * u'
            sol := localsolve(sys + scalarMatrix(n, p), inum, v)
            ratsform := ratsform + integralRepresents(sol, v ** (j::N))
            inum := [(qelt(inum, i) - p * qelt(sol, i) -
                      dot(row(sys, i - nn), sol))
                      exquo v]::UP - u * derivation qelt(sol, i)
        for i in minIndex inum .. maxIndex inum
            iden := u * v
    [ratsform, integralRepresents(inum, iden)]

localsolve(mat, vec, modulus) ==
    ans := new(nrows mat, 0)
    diagonal? mat =>
        for i in minIndex ans .. maxIndex ans
            for j in minRowIndex mat .. maxRowIndex mat
                for k in minColIndex mat .. maxColIndex mat repeat
                    (bc := extendedEuclidean(qelt(mat, j, k), modulus,
                      qelt(vec, i))) case "failed" => return new(0, 0)
                    qsetelt_!(ans, i, bc.coef1)
    ans
    sol := particularSolution(
        map(x+->x::RF, mat)$MatrixCategoryFunctions2(UP, Vector UP, Vector UP, Matrix UP, RF,
Vector RF, Vector RF, Matrix RF),
map(x\rightarrow x::RF, vec)$VectorFunctions2$(UP, RF))$LinearSystemMatrixPackage(RF, Vector RF, Vector RF, Matrix RF)
sol case "failed" => new(0, 0)
for i in minIndex ans .. maxIndex ans repeat
  (bc := extendedEuclidean(denom qelt(sol, i), modulus, 1))
  case "failed" => return new(0, 0)
  qsetelt_!(ans, i, (numer qelt(sol, i) * bc.coef1) rem modulus)
ans

package INTALG AlgebraicIntegrate

— AlgebraicIntegrate.input —

)set break resume
)sys rm -f AlgebraicIntegrate.output
)spool AlgebraicIntegrate.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicIntegrate
--E 1

)spool
)lisp (bye)

— AlgebraicIntegrate.help —

====================================================================
AlgebraicIntegrate examples
====================================================================

This package provides functions for integrating a function on an algebraic curve.

See Also:
o )show AlgebraicIntegrate

AlgebraicIntegrate (INTALG)

Exports:
algintegrate palginfieldint palgintegrate

--- package INTALG AlgebraicIntegrate ---

)abbrev package INTALG AlgebraicIntegrate
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 19 May 1993
++ Description:
++ This package provides functions for integrating a function
++ on an algebraic curve.

AlgebraicIntegrate(R0, F, UP, UPUP, R): Exports == Implementation where
R0 : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
F : Join(AlgebraicallyClosedField, FunctionSpace R0)
UP : UnivariatePolynomialCategory F
UPUP : UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F, UP, UPUP)
SE ==> Symbol
Z ==> Integer
Q ==> Fraction Z
SUP ==> SparseUnivariatePolynomial F
QF ==> Fraction UP
GP ==> LaurentPolynomial(F, UP)
K ==> Kernel F
IR ==> IntegrationResult R
UPQ ==> SparseUnivariatePolynomial Q
UPR ==> SparseUnivariatePolynomial R
FRQ ==> Factored UPQ
FD ==> FiniteDivisor(F, UP, UPUP, R)
FAC ==> Record(factor:UPQ, exponent:Z)
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
DIV ==> Record(num:R, den:UP, derivden:UP, gd:UP)
FAIL0 ==> error "integrate: implementation incomplete (constant residues)"
FAIL1 ==> error "integrate: implementation incomplete (non-algebraic residues)"
FAIL2 ==> error "integrate: implementation incomplete (residue poly has multiple non-linear factors)"
FAIL3 ==> error "integrate: implementation incomplete (has polynomial part)"
NOTI ==> error "Not integrable (provided residues have no relations)"

Exports ==> with
algintegrate : (R, UP -> UP) -> IR
  ++ algintegrate(f, d) integrates f with respect to the derivation d.
palgintegrate : (R, UP -> UP) -> IR
  ++ palgintegrate(f, d) integrates f with respect to the derivation d.
  ++ Argument f must be a pure algebraic function.
palinfieldint: (R, UP -> UP) -> Union(R, "failed")
  ++ palginfieldint(f, d) returns an algebraic function g
  ++ such that \pad{dg = f} if such a g exists, "failed" otherwise.
  ++ Argument f must be a pure algebraic function.
Implementation ==> add
import FD
import DoubleResultantPackage(F, UP, UPUP, R)
import PointsOfFiniteOrder(RO, F, UP, UPUP, R)
import AlgebraicHermiteIntegration(F, UP, UPUP, R)
import InnerCommonDenominator(Z, Q, List Z, List Q)
import FunctionSpaceUnivariatePolynomialFactor(RO, F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, RO, SparseMultivariatePolynomial(RO, K), F)
F2R : F -> R
F2UPR : F -> UPR
UP2SUP : UP -> SUP
SUP2UP : SUP -> UP
UPQ2F : UPQ -> UP
univ : (F, K) -> QF
pLogDeriv : (LOG, R -> R) -> R
nonLinear : List FAC -> Union(FAC, "failed")
PACKAGE INTALG ALGEBRAICINTEGRATE

mkLog  : (UP, Q, R, F) -> List LOG
R2UP   : (R, K) -> UPR
alglogint : (R, UP -> UP) -> Union(List LOG, "failed")
palglogint : (R, UP -> UP) -> Union(List LOG, "failed")
trace00 : (DIV, UP, List LOG) -> Union(List LOG, "failed")
trace0 : (DIV, UP, Q, FD) -> Union(List LOG, "failed")
trace1 : (DIV, UP, List Q, List FD, Q) -> Union(List LOG, "failed")
nonQ   : (DIV, UP) -> Union(List LOG, "failed")
rlift  : (F, K, K) -> R
varRoot? : (UP, F -> F) -> Boolean
algintexp : (R, UP -> UP) -> IR
algintprim : (R, UP -> UP) -> IR

dummy:R := 0

dumx := kernel(new()$SE)$K
dumy := kernel(new()$SE)$K

F2UPR f == F2R(f::UPR)
F2R f == f::UP::QF::R

algintexp(f, derivation) ==
  d := (c := integralCoordinates f).den
  v := c.num
  vp:Vector(GP) := new(n := #v, 0)
  vf:Vector(QF) := new(n, 0)
  for i in minIndex v .. maxIndex v repeat
    r := separate(qelt(v, i) / d)$GP
    qsetelt_!(vf, i, r.fracPart)
    qsetelt_!(vp, i, r.polyPart)
  ff := represents(vf, w := integralBasis())
  h := HermiteIntegrate(ff, derivation)
  p := represents(       map((x1:GP):QF+->convert(x1)@QF, vp)$VectorFunctions2(GP, QF), w)
  zero?(h.logpart) and zero? p => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
    mkAnswer(h.answer, empty(), [[p + h.logpart, dummy]])
  zero? p => mkAnswer(h.answer, u::List(LOG), empty())
FAIL3

algintprim(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
    mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
    mkAnswer(h.answer, u::List(LOG), empty())
FAIL3

-- checks whether f = +/\[ci (ui)'/(ui)]
-- f dx must have no pole at infinity
palglogint(f, derivation) ==
rec := algSplitSimple(f, derivation)
ground?(r := doubleResultant(f, derivation)) => "failed"
-- r(z) has roots which are the residues of f at all its poles
(u := qfactor r) case "failed" => nonQ(rec, r)
(fc := nonLinear(lf := factors(u::FRQ))) case "failed" => FAIL2
-- at this point r(z) = fc(z) (z - b1)^e1 .. (z - bk)^ek
-- where the ri's are rational numbers, and fc(z) is arbitrary
-- (fc can be linear too)
-- la = [b1,...,bk] (all rational residues)
la := [- coefficient(q.factor, 0) for q in remove!(fc::FAC, lf)]
-- ld = [D1,...,Dk] where Di is the sum of places where f has residue bi
ld := [divisor(rec.num, rec.den, rec.derivden, rec.gd, b::F) for b in la]
pp := UPQ2F(fc.factor)
-- bb = - sum of all the roots of fc (i.e. the other residues)
zero?(bb := coefficient(fc.factor, (degree(fc.factor) - 1)::NonNegativeInteger)) =>
-- cd = [[a1,...,ak], d] such that bi = ai/d
cd := splitDenominator la
-- g = gcd(a1,...,ak), so bi = (g/d) ci with ci = bi / g
-- so [g/d] is a basis for [a1,...,ak] over the integers
-- of f except the ones which are root of fc(z)
g := gcd(cd.num)
dv0 := d/[(a quo g) * dv for a in cd.num for dv in ld]
trace0(rec, pp, g / cd.den, dv0)
trace1(rec, pp, la, ld, bb)

UPQ2F p ==
map((x:Q):F+->x::F,p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

UP2SUP p ==
map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, UP, F, SUP)

SUP2UP p ==
map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, SUP, F, UP)

varRoot?(p, derivation) ==
for c in coefficients primitivePart p repeat
  derivation(c) ^= 0 => return true
false

pLogDeriv(log, derivation) ==
map(derivation, log.coeff) ^= 0 =>
  error "can only handle logs with constant coefficients"
-- one?(n := degree(log.coeff)) =>
((n := degree(log.coeff)) = 1) =>
c := - (leadingCoefficient reductum log.coeff)
  / (leadingCoefficient log.coeff)
ans := (log.logand) c
(log.scalar)::R * c * derivation(ans) / ans
numlog := map(derivation, log.logand)
(diflog := extendedEuclidean(log.logand, log.coeff, numlog)) case
  "failed" => error "this shouldn't happen"
algans := diflog.coef1
ans:=R := 0
for i in 0..n-1 repeat
  algans := (algans * monomial(1, 1)) rem log.coeff
  ans := ans + coefficient(algans, i)
(log.scalar)::R * ans

R2UP(f, k) ==
x := dumx :: F
g :=
  (map((f1:QF):F+>f1(x), lift f)_
    $UnivariatePolynomialCategoryFunctions2(QF,UPUP,F,UP))
  (y := dumy::F)_
  map((x1:F):R+>rlift(x1, dumx, dumy), univariate(g, k, minPoly k))_
    $UnivariatePolynomialCategoryFunctions2(F,SUP,R,UPR)

univ(f, k) ==
g := univariate(f, k)
(SUP2UP numer g) / (SUP2UP denom g)

rlift(f, kx, ky) ==
  reduce map(x1+>univ(x1, kx), retract(univariate(f, ky))@SUP)_
    $UnivariatePolynomialCategoryFunctions2(F,SUP,QF,UPUP)

nonQ(rec, p) ==
  empty? rest(lf := factors ffactor primitivePart p) =>
    trace00(rec, first(lf).factor, empty()$List(LOG))
  FAIL1

-- case when the irreducible factor p has roots which sum to 0
-- p is assumed doubly transitive for now
trace0(rec, q, r, dv0) ==
  lg:List(LOG) :=
    zero? dv0 => empty()
  (rc0 := torsionIfCan dv0) case "failed" => NOTI
  mkLog(1, r / (rc0.order::Q), rc0.function, 1)
  trace00(rec, q, lg)

trace00(rec, pp, lg) ==
p0 := divisor(rec.num, rec.den, rec.derivden, rec.gd,
  alpha0 := zero0f UP2SUP pp)
q := (pp exquo (monomial(1, 1)$UP - alpha0::UP))::UP
alpha := root0f UP2SUP q
dvr := divisor(rec.num, rec.den, rec.derivden, rec.gd, alpha) - p0
(rc := torsionIfCan dvr) case "failed" =>
  degree(pp) <= 2 => "failed"
concat(lg, mkLog(q, inv(rc.order::Q), rc.function, alpha))

-- case when the irreducible factor p has roots which sum <> 0
-- the residues of f are of the form [a1,...,ak] rational numbers
-- plus all the roots of q(z), which is squarefree
-- la is the list of residues la := [a1,...,ak]
-- ld is the list of divisors [D1,...,Dk] where Di is the sum of all the
-- places where f has residue ai
-- q(z) is assumed doubly transitive for now.
-- let [alpha_1,...,alpha_m] be the roots of q(z)
-- in this function, b = - alpha_1 - ... - alpha_m is <> 0
-- which implies only one generic log term
-- trace1(rec, q, la, ld, b) ==
-- cd = [[b1,...,bk], d] such that ai / b = bi / d
-- then, a basis for all the residues of f over the integers is
-- [beta_1 = - alpha_1 / d,..., beta_m = - alpha_m / d], since:
-- alpha_i = - d beta_i
-- ai = (ai / b) * b = (bi / d) * b = b1 * beta_1 + ... + bm * beta_m
-- linear independence is a consequence of the doubly transitive assumption
-- v0 is the divisor +(bi Di) corresponding to the residues [a1,...,ak]
-- v0 := +(a * dv for a in cd.num for dv in ld]
-- alpha is a generic root of q(z)
-- alpha := rootOf UP2SUP q
-- v is the divisor corresponding to all the residues
-- v := v0 - cd.den * divisor(rec.num, rec.den, rec.derivden, rec.gd, alpha)
-- degree(q) <= 2 => "failed" -- guaranteed doubly-transitive
-- NOTI -- maybe doubly-transitive
-- mkLog(q, inv((- rc.order * cd.den)::Q), rc.function, alpha)

mkLog(q, scalr, lgd, alpha) ==
  degree(q) <= 1 =>
  [[scalr, monomial(1, 1)$UPR - F2UPR alpha, lgd::UPR]]
  [[scalr, map(F2R, q)$UnivariatePolynomialCategoryFunctions2(F,UP,R,UPR),
    R2UP(lgd, retract(alpha)@K)]]

-- return the non-linear factor, if unique
-- or any linear factor if they are all linear
nonLinear l ==
  found:Boolean := false
  ans := first l
  for q in l repeat
    if degree(q.factor) > 1 then
      found => return "failed"
      found := true
      ans := q
-- f dx must be locally integral at infinity
palginfieldint(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer
  "failed"

-- f dx must be locally integral at infinity
palgintegrate(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer::IR
  (not integralAtInfinity?(h.logpart)) or
  ((u := palglogint(h.logpart, derivation)) case "failed") =>
    mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
  zero?(difFirstKind := h.logpart - +/\[pLogDeriv(lg, x1+- differentiate(x1, derivation)) for lg in u::List(LOG)\]) =>
    mkAnswer(h.answer, u::List(LOG), empty())
  mkAnswer(h.answer, u::List(LOG), [[difFirstKind, dummy]])

-- for mixed functions. f dx not assumed locally integral at infinity
algintegrate(f, derivation) ==
  zero? degree(x' := derivation(x := monomial(1, 1)$UP)) =>
    algintprim(f, derivation)
  ((xx := x' exquo x) case UP) and
  (retractIfCan(xx::UP)$Union(F, "failed") case F) =>
    algintexp(f, derivation)
  error "should not happen"

alglogint(f, derivation) ==
  varRoot?(doubleResultant(f, derivation),
    x1+-retract(derivation(x1::UP)$F) => "failed"
FAIL0

— INTALG.dotabb —

"INTALG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTALG"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"INTALG" -> "ACF"
"INTALG" -> "FS"
"INTALG" -> "FFCAT"
package INTAF AlgebraicIntegration

--- AlgebraicIntegration.input ---

)set break resume
)sys rm -f AlgebraicIntegration.output
)spool AlgebraicIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicIntegration
--E 1

)spool
)lisp (bye)

--- AlgebraicIntegration.help ---

====================================================================
AlgebraicIntegration examples
====================================================================

This package provides functions for the integration of algebraic integrands over transcendental functions;

See Also:
  o )show AlgebraicIntegration

---
AlgebraicIntegration (INTAF)

Exports:
algint

--- package INTAF AlgebraicIntegration ---

)abbrev package INTAF AlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 12 October 1988
++ Date Last Updated: 4 June 1988
++ Description:
++ This package provides functions for the integration of
++ algebraic integrands over transcendental functions;

AlgebraicIntegration(R, F):Exports == Implementation where
  R : Join(OrderedSet, IntegralDomain)
  F : Join(AlgebraicallyClosedField, FunctionSpace R)

  SY ==> Symbol
  N ==> NonNegativeInteger
  K ==> Kernel F
  P ==> SparseMultivariatePolynomial(R, K)
  UP ==> SparseUnivariatePolynomial F
  RF ==> Fraction UP
  UPUP==> SparseUnivariatePolynomial RF
  IR ==> IntegrationResult F
  IR2 ==> IntegrationResultFunctions2(curve, F)
  ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
  FAIL==> error "failed - cannot handle that integrand"

Exports == with
  algint: (F, K, K, UP -> UP) -> IR
    ++ algint(f, x, y, d) returns the integral of \spad{f(x,y)dx}
    ++ where y is an algebraic function of x;
    ++ d is the derivation to use on \spad{k[x]}. 
Implementation \impl add

\begin{verbatim}
import ChangeOfVariable(F, UP, UPUP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

rootintegrate: (F, K, K, UP -> UP) -> IR
algintegrate : (F, K, K, UP -> UP) -> IR
UPUP2F : (UPUP, RF, K, K) -> F
F2UPUP : (F, K, K, UP) -> UPUP
UP2UPUP : (UP, K) -> UPUP

F2UPUP(f, kx, k, p) == UP2UPUP(univariate(f, k, p), kx)

rootintegrate(f, t, k, derivation) ==
  r1 := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
  f1 := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
  r := radPoly(r1.poly)::Record(radicand:RF, deg:N)
  q := retract(r.radicand)
  curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
  map(x1+->UPUP2F(lift x1, r1.coef, t, k),
    algintegrate(reduce f1, derivation)$ALG)$IR2

algintegrate(f, t, k, derivation) ==
  r1 := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
  f1 := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
  modulus := UP2UPUP(p := minPoly k, t)
  curve := AlgebraicFunctionField(F, UP, UPUP, r1.poly)
  map(x1+->UPUP2F(lift x1, r1.coef, t, k),
    algintegrate(reduce f1, derivation)$ALG)$IR2

UP2UPUP(p, k) ==
  map(x1+->univariate(x1,k),p)$SparseUnivariatePolynomialFunctions2(F,RF)

UPUP2F(p, cf, t, k) ==
  map((x1:RF):F+->multivariate(x1, t),
    p)$SparseUnivariatePolynomialFunctions2(RF, F)
    (multivariate(cf, t) * k::F)

algint(f, t, y, derivation) ==
  is?(y, "nthRoot":SY) => rootintegrate(f, t, y, derivation)
  is?(y, "rootOf":SY) => algintegrate(f, t, y, derivation)
  FAIL

— INTAF.dotabb —

"INTAF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTAF"]
package ALGMANIP AlgebraicManipulations

---

--S 1 of 1
>show AlgebraicManipulations
--E 1

---

---

AlgebraicManipulations examples

AlgebraicManipulations provides functions to simplify and expand expressions involving algebraic operators.

See Also:
  o )show AlgebraicManipulations

---
AlgebraicManipulations (ALGMANIP)

Exports:
ratDenom  ratPoly  rootKerSimp  rootPower
rootProduct  rootSimp  rootSplit

— package ALGMANIP AlgebraicManipulations —

)abbrev package ALGMANIP AlgebraicManipulations
++ Author: Manuel Bronstein
++ Date Created: 28 Mar 1988
++ Date Last Updated: 5 August 1993
++ Description:
++ AlgebraicManipulations provides functions to simplify and expand
++ expressions involving algebraic operators.

AlgebraicManipulations(R, F): Exports == Implementation where
R : IntegralDomain
F : Join(Field, ExpressionSpace) with
   numer  : $ -> SparseMultivariatePolynomial(R, Kernel $)
      ++ numer(x) \undocumented
   denom  : $ -> SparseMultivariatePolynomial(R, Kernel $)
      ++ denom(x) \undocumented
   coerce: SparseMultivariatePolynomial(R, Kernel $) -> $
      ++ coerce(x) \undocumented

N ==> NonNegativeInteger
Z ==> Integer
OP ==> BasicOperator
SY ==> Symbol
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
RF ==> Fraction P
REC ==> Record(ker:List K, exponent: List Z)
ALGOP ==> "%alg"
NTHR ==> "nthRoot"
Exports ==> with

rootSplit: F -> F
  ++ rootSplit(f) transforms every radical of the form
  ++ \spad{(a/b)**(1/n)} appearing in f into \spad{a**(1/n) / b**(1/n)}.
  ++ This transformation is not in general valid for all
  ++ complex numbers \spad{a} and b.
ratDenom : F -> F
  ++ ratDenom(f) rationalizes the denominators appearing in f
  ++ by moving all the algebraic quantities into the numerators.
ratDenom : (F, F) -> F
  ++ ratDenom(f, a) removes \spad{a} from the denominators in f
  ++ if \spad{a} is an algebraic kernel.
ratDenom : (F, List F) -> F
  ++ ratDenom(f, [a1,...,an]) removes the ai’s which are
  ++ algebraic kernels from the denominators in f.
ratDenom : (F, List K) -> F
  ++ ratDenom(f, [a1,...,an]) removes the ai’s which are
  ++ algebraic from the denominators in f.
ratPoly : F -> SparseUnivariatePolynomial F
  ++ ratPoly(f) returns a polynomial p such that p has no
  ++ algebraic coefficients, and \spad{p(f) = 0}.
if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
and F has FunctionSpace(R) then
  rootPower : F -> F
  ++ rootPower(f) transforms every radical power of the form
  ++ \spad{(a**(1/n))**m} into a simpler form if \spad{m} and
  ++ \spad{n} have a common factor.
rootProduct: F -> F
  ++ rootProduct(f) combines every product of the form
  ++ \spad{(a**(1/n))**m * (a**(1/s))**t} into a single power
  ++ of a root of \spad{a}, and transforms every radical power
  ++ of the form \spad{(a**(1/n))**m} into a simpler form.
rootSimp : F -> F
  ++ rootSimp(f) transforms every radical of the form
  ++ \spad{(a * b**(q*n+r))**(1/n)} appearing in f into
  ++ \spad{a**(1/n) * (a * b**r)**(1/n)}.
  ++ This transformation is not in general valid for all
  ++ complex numbers b.
rootKerSimp: (OP, F, N) -> F
  ++ rootKerSimp(op,f,n) should be local but conditional.

Implementation ==> add

innerRF : (F, List K) -> F
rootExpand : K -> F
algkernels : List K -> List K
rootkernels: List K -> List K

dummy := kernel(new()$SY)$K
ratDenom x == innerRF(x, algkernels tower x)
ratDenom(x:F, l:List K):F == innerRF(x, algkernels l)
ratDenom(x:F, y:F) == ratDenom(x, [y])
ratDenom(x:F, l:List F) == ratDenom(x, [retract(y)@K for y in l]$List(K))
algkernels l == select_!((z1:K):Boolean +-> has?(operator z1, ALGOP), l)
rootkernels l == select_!((z1:K):Boolean +-> is?(operator z1, NTHR::SY), l)

ratPoly x ==
  numer univariate(denom(ratDenom inv(dummy::P::F - x))::F, dummy)

rootSplit x ==
  lk := rootkernels tower x
  eval(x, lk, [rootExpand k for k in lk])

rootExpand k ==
  x := first argument k
  n := second argument k
  op := operator k
  op(numer(x)::F, n) / op(denom(x)::F, n)

-- all the kernels in ll must be algebraic
innerRF(x, ll) ==
  empty?(l := sort_!((z1:K,z2:K):Boolean +-> z1 > z2, kernels x)$List(K)) or
  empty? setIntersection(ll, tower x) => x
  lk := empty()$List(K)
  while not member?(k := first l, ll) repeat
    lk := concat(k, lk)
    empty?(l := rest l) =>
    return eval(x, lk, [map((z3:F):F+->innerRF(z3, ll), kk) for kk in lk])
  q := univariate(eval(x, lk,
      [map((z4:F):F+->innerRF(z4, ll), kk) for kk in lk]),k,minPoly k)
  map((z5:F):F+->innerRF(z5, ll), q) (map((z6:F):F+->innerRF(z6, ll), k))

if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
  and F has FunctionSpace(R) then
  import PolynomialRoots(IndexedExponents K, K, R, P, F)
sroot : K -> F
sinroot : (OP, F, N) -> F
radeval: (P, K) -> F
breakup: List K -> List REC

if R has RadicalCategory then
  rootKerSimp(op, x, n) ==
    (r := retractIfCan(x)@Union(R, "failed")) case R =>
      nthRoot(r::R, n)::F
      inroot(op, x, n)
else
  rootKerSimp(op, x, n) == inroot(op, x, n)
-- l is a list of nth-roots, returns a list of records of the form
-- [[a**(1/n1),a**(1/n2),...],[n1,n2,...]]
-- such that the whole list covers l exactly
breakup l ==
  empty? l => empty()
  k := first l
  a := first(arg := argument(k := first l))
  n := retract(second arg)@Z
  expo := empty()$List(Z)
  others := same := empty()$List(K)
  for kk in rest l repeat
    if (a = first(arg := argument kk)) then
      same := concat(kk, same)
      expo := concat(retract(second arg)@Z, expo)
    else others := concat(kk, others)
  ll := breakup others
  concat([concat(k, same), concat(n, expo)], ll)

rootProduct x ==
  for rec in breakup rootkernels tower x repeat
    k0 := first(l := rec.ker)
    nx := numer x; dx := denom x
    if empty? rest l then x := radeval(nx, k0) / radeval(dx, k0)
    else
      n := lcm(rec.exponent)
      k := kernel(operator k0, [first argument k0, n::F], height k0)$K
      lv := [monomial(1, k, (n quo m)::N) for m in rec.exponent]$List(P)
      x := radeval(eval(nx, l, lv), k) / radeval(eval(dx, l, lv), k)
  x

rootPower x ==
  for k in rootkernels tower x repeat
    x := radeval(numer x, k) / radeval(denom x, k)
  x

-- replaces (a**(1/n))**m in p by a power of a simpler radical of a if
-- n and m have a common factor
radeval(p, k) ==
  a := first(arg := argument k)
  n := (retract(second arg)@Integer):NonNegativeInteger
  ans:F := 0
  q := univariate(p, k)
  while (d := degree q) > 0 repeat
    term :=
      -- one?(g := gcd(d, n)) => monomial(1, k, d)
      ((g := gcd(d, n)) = 1 => monomial(1, k, d)
       monomial(1, kernel(operator k, [a,(n quo g)::F], height k), d quo g)
       ans := ans + leadingCoefficient(q)::F * term::F
    q := reductum q
leadingCoefficient(q)::F + ans

inroot(op, x, n) ==
  -- one? x => x
  (x = 1) => x
  -- (x ^= -1) and (one?(num := numer x) or (num = -1)) =>
  (x ^= -1) and (((num := numer x) = 1) or (num = -1)) =>
  inv inroot(op, (num * denom x)::F, n)
  (u := isExpt(x, op)) case "failed" => kernel(op, [x, n::F])
  pr := u::Record(var:K, exponent:Integer)
  q := pr.exponent /$Fraction(Z)
  (n * retract(second argument(pr.var))@Z)
  qr := divide(numer q, denom q)
  x := first argument(pr.var)
  x ** qr.quotient * rootKerSimp(op,x,denom(q)::N) ** qr.remainder

sroot k ==
  pr := froot(first(arg := argument k),(retract(second arg)@Z)::N)
  pr.coef * rootKerSimp(operator k, pr.radicand, pr.exponent)

rootSimp x ==
  lk := rootkernels tower x
  eval(x, lk, [sroot k for k in lk])
AlgebraicMultFact examples

This package factors multivariate polynomials over the domain of AlgebraicNumber by allowing the user to specify a list of algebraic numbers generating the particular extension to factor over.

See Also:
o )show AlgebraicMultFact

Exports:
factor
++ Author: P. Gianni
++ Date Created: 1990
++ Description:
++ This package factors multivariate polynomials over the
++ domain of \texttt{AlgebraicNumber} by allowing the user
++ to specify a list of algebraic numbers generating the particular
++ extension to factor over.

\texttt{AlgebraicMultFact(OV,E,P) : C == T}

where

\texttt{AN} => AlgebraicNumber
\texttt{OV} : OrderedSet
\texttt{E} : OrderedAbelianMonoidSup
\texttt{P} => PolynomialCategory(AN,E,OV)
\texttt{BP} => SparseUnivariatePolynomial AN
\texttt{Z} => Integer
\texttt{MParFact} => Record(irr:P,pow:Z)
\texttt{USP} => SparseUnivariatePolynomial P
\texttt{SUParFact} => Record(irr:USP,pow:Z)
\texttt{SUPFinalFact} => Record(contp:R,factors:List SUParFact)
\texttt{MFinalFact} => Record(contp:R,factors:List MParFact)

\texttt{C == with}
\texttt{factor : (P,L AN) -> Factored P}
++ \texttt{factor(p,lan)} factors the polynomial \texttt{p} over the extension
++ generated by the algebraic numbers given by the list \texttt{lan}.
\texttt{factor : (USP,L AN) -> Factored USP}
++ \texttt{factor(p,lan)} factors the polynomial \texttt{p} over the extension
++ generated by the algebraic numbers given by the list \texttt{lan}.
++ \texttt{p} is presented as a univariate polynomial with multivariate
++ coefficients.

\texttt{T == add}

\texttt{AF := AlgFactor(BP)}

\texttt{INNER => InnerMultFact(OV,E,AN,P)}

\texttt{factor(p:P,lalg:L AN) : Factored P ==}
++ \texttt{factor(p,lalg)} => \texttt{factor(p,lan)}$AF$\texttt{INNER}

\texttt{factor(up:USP,lalg:L AN) : Factored USP ==}
++ \texttt{factor(up,lalg)} => \texttt{factor(p,lalg)}$AF$\texttt{INNER}
package ALGPKG AlgebraPackage

-- AlgebraPackage.input --

)set break resume
)sys rm -f AlgebraPackage.output
)spool AlgebraPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraPackage
--E 1

)spool
)lisp (bye)

-----

--- AlgebraPackage.help ---

====================================================================
AlgebraPackage examples
====================================================================

AlgebraPackage assembles a variety of useful functions for general algebras.

See Also:
o )show AlgebraPackage

-----
AlgebraPackage (ALGPKG)

Exports:

basis basisOfCenter basisOfCentroid basisOfCommutingElements
basisOfLeftNucleus basisOfLeftNucleoid basisOfMiddleNucleus basisOfNucleus
basisOfRightNucleus basisOfRightNucleoid biRank doubleRank
radicalOfLeftTraceForm rightRank weakBiRank

— package ALGPKG AlgebraPackage —

\textit{AlgebraPackage}(R: \text{IntegralDomain}, \ A: \text{FramedNonAssociativeAlgebra}(R)): _  
\text{public} \Rightarrow \text{private} \text{ where}

V \Rightarrow \text{Vector}  
M \Rightarrow \text{Matrix}  
I \Rightarrow \text{Integer}  
\text{NNI} \Rightarrow \text{NonNegativeInteger}  
\text{REC} \Rightarrow \text{Record}(\text{particular: Union}(V R, "failed"), \text{basis: List} \ V R)
LSMP ==> LinearSystemMatrixPackage(R,V,R,V,M,R)

public ==> with

leftRank: A -> NonNegativeInteger
   ++ leftRank(x) determines the number of linearly independent elements
   ++ in \spad{x*b1},...,\spad{x*bn},
   ++ where \spad{b=[b1,...,bn]} is a basis.
rightRank: A -> NonNegativeInteger
   ++ rightRank(x) determines the number of linearly independent elements
   ++ in \spad{b1*x},...,\spad{bn*x},
   ++ where \spad{b=[b1,...,bn]} is a basis.
doubleRank: A -> NonNegativeInteger
   ++ doubleRank(x) determines the number of linearly
   ++ independent elements
   ++ in \spad{b1*x},...,\spad{x*bn},
   ++ where \spad{b=[b1,...,bn]} is a basis.
weakBiRank: A -> NonNegativeInteger
   ++ weakBiRank(x) determines the number of
   ++ linearly independent elements
   ++ in the \spad{bi*x*bj}, \spad{i,j=1,...,n},
   ++ where \spad{b=[b1,...,bn]} is a basis.
biRank: A -> NonNegativeInteger
   ++ biRank(x) determines the number of linearly independent elements
   ++ in \spad{x}, \spad{x*bi}, \spad{bi*x}, \spad{bi*x*bj},
   ++ \spad{i,j=1,...,n},
   ++ where \spad{b=[b1,...,bn]} is a basis.
   ++ Note that if \spad{A} has a unit,
   ++ then doubleRank, weakBiRank, and biRank coincide.
basisOfCommutingElements: () -> List A
   ++ basisOfCommutingElements() returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = commutator(x,a)} for all
   ++ \spad{a} in \spad{A}.
basisOfLeftAnnihilator: A -> List A
   ++ basisOfLeftAnnihilator(a) returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = x*a}.
basisOfRightAnnihilator: A -> List A
   ++ basisOfRightAnnihilator(a) returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = a*x}.
basisOfLeftNucleus: () -> List A
   ++ basisOfRightNucleus() returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = associator(x,a,b)}
   ++ for all \spad{a,b} in \spad{A}.
basisOfRightNucleus: () -> List A
   ++ basisOfRightNucleus() returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = associator(a,b,x)}
   ++ for all \spad{a,b} in \spad{A}.
basisOfMiddleNucleus: () -> List A
   ++ basisOfMiddleNucleus() returns a basis of the space of
   ++ all x of \spad{A} satisfying \spad{0 = associator(a,x,b)}
++ for all \spad{a}, b in \spad{A}.

\textbf{basisOfNucleus: () -> List A}
++ basisOfNucleus() returns a basis of the space of
++ all \spad{x} of \spad{A} satisfying
++ \spad{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
++ for all \spad{a}, \spad{b} in \spad{A}.

\textbf{basisOfCenter: () -> List A}
++ basisOfCenter() returns a basis of the space of
++ all \spad{x} of \spad{A} satisfying \spad{comutator(x,a) = 0} and
++ \spad{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
++ for all \spad{a}, \spad{b} in \spad{A}.

\textbf{basisOfLeftNucloid: () -> List Matrix R}
++ basisOfLeftNucloid() returns a basis of the space of
++ endomorphisms of \spad{A} as right module.
++ Note that left nucloid coincides with left nucleus
++ if \spad{A} has a unit.

\textbf{basisOfRightNucloid: () -> List Matrix R}
++ basisOfRightNucloid() returns a basis of the space of
++ endomorphisms of \spad{A} as left module.
++ Note that right nucloid coincides with right nucleus
++ if \spad{A} has a unit.

\textbf{basisOfCentroid: () -> List Matrix R}
++ basisOfCentroid() returns a basis of the centroid, i.e. the
++ endomorphism ring of \spad{A} considered as \spad{(A,A)}-bimodule.

\textbf{radicalOfLeftTraceForm: () -> List A}
++ radicalOfLeftTraceForm() returns basis for null space of
++ \spad{leftTraceMatrix()}, if the algebra is
++ associative, alternative or a Jordan algebra, then this
++ space equals the radical (maximal nil ideal) of the algebra.

if \spad{R} has EuclideanDomain then
  \textbf{basis : V A -> V A}
  ++ basis(va) selects a basis from the elements of va.

private => add

-- constants

\textbf{n} : PositiveInteger := rank()$A
\textbf{n2} : PositiveInteger := n*n
\textbf{n3} : PositiveInteger := n*n2
\textbf{gamma} : Vector Matrix R := structuralConstants()$A

-- local functions

\textbf{convVM : Vector R -> Matrix R}
  -- converts n2-vector to (n,n)-matrix row by row
\textbf{convMV : Matrix R -> Vector R}
  -- converts n-square matrix to n2-vector row by row
convVM v ==
cond : Matrix(R) := new(n,n,0$R)$M(R)
z : Integer := 0
for i in 1..n repeat
  for j in 1..n repeat
    z := z+1
    setelt(cond,i,j,v.z)
cond

-- convMV m ==
-- vec : Vector(R) := new(n*n,0$R)
-- z : Integer := 0
-- for i in 1..n repeat
--  for j in 1..n repeat
--    z := z+1
--    setelt(vec,z,elt(m,i,j))
-- vec

radicalOfLeftTraceForm() ==
ma : M R := leftTraceMatrix()$A
map(represents, nullSpace ma)$ListFunctions2(Vector R, A)

basisOfLeftAnnihilator a ==
ca : M R := transpose (coordinates(a) :: M R)
cond : M R := reduce(vertConcat$(M R),
    [ca*transpose(gamma.i) for i in 1..#gamma])
map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfRightAnnihilator a ==
ca : M R := transpose (coordinates(a) :: M R)
cond : M R := reduce(vertConcat$(M R),
    [ca*(gamma.i) for i in 1..#gamma])
map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfLeftNucloid() ==
cond : Matrix(R) := new(n3,n2,0$R)$M(R)
condo: Matrix(R) := new(n3,n2,0$R)$M(R)
z : Integer := 0
for i in 1..n repeat
  for j in 1..n repeat
    r1 : Integer := 0
    for k in 1..n repeat
      z := z + 1
      -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
r2 : Integer := i
    for r in 1..n repeat
      r1 := r1 + 1
-- here r1 equals (k-1)*n+r  (loop-invariant)
setelt(cond,z,r1,elt(gamma.r,i,j))

-- here r2 equals (r-1)*n+i  (loop-invariant)
setelt(condo,z,r2,-elt(gamma.k,r,j))

r2 := r2 + n

[convVM(sol) for sol in nullSpace(cond+condo)]

basisOfCommutingElements() ==
  --gamma1 := first gamma
  --gamma1 := gamma1 - transpose gamma1
  --cond : Matrix(R) := gamma1 :: Matrix(R)
  --for i in 2..n repeat
  --  gammak := gamma.i
  --  gammak := gammak - transpose gammak
  --  cond := vertConcat(cond, gammak :: Matrix(R))$Matrix(R)
  --map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

  cond : M R := reduce(vertConcat$(M R),
  [(gam := gamma.i) - transpose gam for i in 1..#gamma]
  )
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfLeftNucleus() ==
  condi: Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
    for j in 1..n repeat
      for s in 1..n repeat
        z := z+1
        for i in 1..n repeat
          entry : R := 0
          for l in 1..n repeat
            entry := entry+elt(gamma.l,j,k)*elt(gamma.s,i,l) -
            elt(gamma.l,i,j)*elt(gamma.s,l,k)
          setelt(condi,z,i,entry)$Matrix(R)
  map(represents, nullSpace condi)$ListFunctions2(Vector R, A)

basisOfRightNucleus() ==
  condo : Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
    for j in 1..n repeat
      for s in 1..n repeat
        z := z+1
        for i in 1..n repeat
          entry : R := 0
          for l in 1..n repeat
            entry := entry+elt(gamma.l,k,i)*elt(gamma.s,j,l) -
            elt(gamma.l,j,k)*elt(gamma.s,l,i)
          setelt(condo,z,i,entry)$Matrix(R)
  map(represents, nullSpace condo)$ListFunctions2(Vector R, A)
basisOfMiddleNucleus() ==
  conda : Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
    for j in 1..n repeat
      for s in 1..n repeat
        z := z+1
        for i in 1..n repeat
          entry : R := 0
          for l in 1..n repeat
            entry := entry + elt(gamma.l,j,i)*elt(gamma.s,l,k) - elt(gamma.l,i,k)*elt(gamma.s,j,l)
          setelt(conda,z,i,entry)$Matrix(R)
    map(represents, nullSpace conda)$ListFunctions2(Vector R,A)

basisOfNucleus() ==
  condi: Matrix(R) := new(3*n3,n,0$R)$Matrix(R)
  z : Integer := 0
  u : Integer := n3
  w : Integer := 2*n3
  for k in 1..n repeat
    for j in 1..n repeat
      for s in 1..n repeat
        z := z+1
        u := u+1
        w := w+1
        for i in 1..n repeat
          entry : R := 0
          enter : R := 0
          ent : R := 0
          for l in 1..n repeat
            entry := entry + elt(gamma.l,j,k)*elt(gamma.s,i,l) - elt(gamma.l,i,j)*elt(gamma.s,l,k)
            enter := enter + elt(gamma.l,k,i)*elt(gamma.s,j,l) - elt(gamma.l,j,k)*elt(gamma.s,l,i)
            ent := ent + elt(gamma.l,j,k)*elt(gamma.s,i,l) - elt(gamma.l,j,i)*elt(gamma.s,l,k)
          setelt(condi,z,i,entry)$Matrix(R)
          setelt(condi,u,i,enter)$Matrix(R)
          setelt(condi,w,i,ent)$Matrix(R)
    map(represents, nullSpace condi)$ListFunctions2(Vector R,A)

basisOfCenter() ==
  gamma1 := first gamma
  gamma1 := gamma1 - transpose gamma1
  cond : Matrix(R) := gamma1 :: Matrix(R)
  for i in 2..n repeat
    gammak := gamma.i
gammak := gammak - transpose gammak
cond := vertConcat(cond, gammak :: Matrix(R))$Matrix(R)
B := cond :: Matrix(R)
condi: Matrix(R) := new(2*n3,n,0$R)$Matrix(R)
z : Integer := 0
u : Integer := n3
for k in 1..n repeat
  for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
      u := u+1
      for i in 1..n repeat
        entry : R := 0
        enter : R := 0
        for l in 1..n repeat
          entry := entry + elt(gamma.l,j,k)*elt(gamma.s,i,l) 
          - elt(gamma.l,i,j)*elt(gamma.s,l,k)
          enter := enter + elt(gamma.l,k,i)*elt(gamma.s,j,l) 
          - elt(gamma.l,j,k)*elt(gamma.s,l,i)
        setelt(condi,z,i,entry)$Matrix(R)
        setelt(condi,u,i,enter)$Matrix(R)
    D := vertConcat(condi,B)$Matrix(R)
    map(represents, nullSpace D)$ListFunctions2(Vector R, A)

basisOfRightNucloid() ==
cond : Matrix(R) := new(n3,n2,0$R)$M(R)
condo: Matrix(R) := new(n3,n2,0$R)$M(R)
z : Integer := 0
for i in 1..n repeat
  for j in 1..n repeat
    r1 : Integer := 0
    for k in 1..n repeat
      z := z + 1
      -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
    r2 : Integer := i
    for r in 1..n repeat
      r1 := r1 + 1
      -- here r1 equals (k-1)*n+r (loop-invariant)
      setelt(cond,z,r1,elt(gamma.r,j,i))
    -- here r2 equals (r-1)*n+i (loop-invariant)
    setelt(condo,z,r2,-elt(gamma.k,j,r))
    r2 := r2 + n
    [convVM(sol) for sol in nullSpace(cond+condo)]

basisOfCentroid() ==
cond : Matrix(R) := new(2*n3,n2,0$R)$M(R)
condo: Matrix(R) := new(2*n3,n2,0$R)$M(R)
z : Integer := 0
u : Integer := n3
for i in 1..n repeat
for j in 1..n repeat
  r1 : Integer := 0
  for k in 1..n repeat
    z := z+1
    u := u+1
    -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
    -- u equals n**3 + (i-1)*n*n+(j-1)*n+k (loop-invariant)
  r2 : Integer := i
  for r in 1..n repeat
    r1 := r1 + 1
    -- here r1 equals (k-1)*n+r (loop-invariant)
    setelt(cond,z,r1,elt(gamma.r,i,j))
    setelt(cond,u,r1,elt(gamma.r,j,i))
    -- here r2 equals (r-1)*n+i (loop-invariant)
    setelt(condo,z,r2,-elt(gamma.k,r,j))
    setelt(condo,u,r2,-elt(gamma.k,j,r))
    r2 := r2 + n
[convVM(sol) for sol in nullSpace(cond+condo)]

doubleRank x ==
  cond : Matrix(R) := new(2*n,n,0$R)
  for k in 1..n repeat
    z : Integer := 0
    u : Integer := n
    for j in 1..n repeat
      z := z+1
      u := u+1
      entry : R := 0
      enter : R := 0
      for i in 1..n repeat
        entry := entry + elt(x,i)*elt(gamma.k,j,i)
        enter := enter + elt(x,i)*elt(gamma.k,i,j)
      setelt(cond,z,k,entry)$Matrix(R)
      setelt(cond,u,k,enter)$Matrix(R)
  rank(cond)$Matrix(R)

weakBiRank(x) ==
  cond : Matrix(R) := new(n2,n,0$R)$Matrix(R)
  for i in 1..n repeat
    for j in 1..n repeat
      z := z+1
      for k in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
          for s in 1..n repeat
            entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
          setelt(cond,z,k,entry)$Matrix(R)
        rank(cond)$Matrix(R)
biRank(x) ==
    cond : Matrix(R) := new(n2+2*n+1,n,0$R)$Matrix(R)
    z : Integer := 0
    for j in 1..n repeat
        for i in 1..n repeat
            z := z+1
            for k in 1..n repeat
                entry : R := 0
                for l in 1..n repeat
                    for s in 1..n repeat
                        entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
                setelt(cond,z,k,entry)$Matrix(R)
    u : Integer := n*n
    w : Integer := n*(n+1)
    c := n2 + 2*n + 1
    for j in 1..n repeat
        u := u+1
        w := w+1
        for k in 1..n repeat
            entry : R := 0
            enter : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,i,j)
                enter := enter + elt(x,i)*elt(gamma.k,i,j)
            setelt(cond,u,k,entry)$Matrix(R)
            setelt(cond,w,k,enter)$Matrix(R)
            setelt(cond,c,j, elt(x,j))
    rank(cond)$(M R)

leftRank x ==
    cond : Matrix(R) := new(n,n,0$R)
    for k in 1..n repeat
        for j in 1..n repeat
            entry : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,i,j)
            setelt(cond,j,k,entry)$Matrix(R)
    rank(cond)$(M R)

rightRank x ==
    cond : Matrix(R) := new(n,n,0$R)
    for k in 1..n repeat
        for j in 1..n repeat
            entry : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,j,i)
            setelt(cond,j,k,entry)$Matrix(R)
    rank(cond)$(M R)
if $R$ has EuclideanDomain then

basis $va =$

$v : V A := \text{remove(zero?, va)}(V A)$

$v : V A := \text{removeDuplicates v}$

empty? $v$ $\Rightarrow$ [0$A$

$m : \text{Matrix } R := \text{coerce(coordinates(v.1))}(\text{Matrix } R)$

for $i$ in 2..\text{maxIndex} $v$ repeat

$m := \text{horizConcat(m,coerce(coordinates(v.i))}(\text{Matrix } R)$

m := rowEchelon m

lj : List Integer := []

h : Integer := 1

mRI : Integer := \text{maxRowIndex} m

mCI : Integer := \text{maxColIndex} m

finished? : Boolean := false

j : Integer := 1

while not finished? repeat

not zero? $m(h,j)$ $\Rightarrow$ -- corner found

lj := cons(j,lj)

h := mRI

while zero? $m(h,j)$ repeat $h := h-1$

finished? := ($h = mRI$

if not finished? then $h := h+1$

if $j < mCI$ then

j := j + 1

else

finished? := true

[v.j for j in reverse lj]

---

package ALGFACT AlgFactor

--- AlgFactor.input ---

)set break resume
CHAPTER 2. CHAPTER A

```lisp
)sys rm -f AlgFactor.output
)spool AlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgFactor
--E 1

)spool
)lisp (bye)

-----

— AlgFactor.help —

====================================================================
AlgFactor examples
====================================================================

Factorization of univariate polynomials with coefficients in AlgebraicNumber.

See Also:
  o )show AlgFactor

-----

AlgFactor (ALGFACT)

Exports:
  doublyTransitive?  factor  split
PACKAGE ALGFACT ALGFACTOR

— package ALGFACT AlgFactor —

)abbrev package ALGFACT AlgFactor
++ Author: Manuel Bronstein
++ Description:
++ Factorization of univariate polynomials with coefficients in
++ \texttt{AlgebraicNumber}.

AlgFactor(UP):Exports == Implementation where

\begin{verbatim}
 UP: UnivariatePolynomialCategory AlgebraicNumber
 N ==> NonNegativeInteger
 Z ==> Integer
 Q ==> Fraction Integer
 AN ==> AlgebraicNumber
 K ==> Kernel AN
 UPQ ==> SparseUnivariatePolynomial Q
 SUP ==> SparseUnivariatePolynomial AN
 FR ==> Factored UP

Exports == with
 factor: (UP, List AN) -> FR
 factor(p, [a1,...,an]) returns a prime factorisation of p
 factor: UP -> FR
 factor(p) returns a prime factorisation of p
 split : UP -> FR
 split(p) returns a prime factorisation of p over its splitting field.
 doublyTransitive?: UP -> Boolean
 doublyTransitive?(p) is true if p is irreducible over
 over the field \textit{K} generated by its coefficients, and
 if \texttt{spad}(p(X) / (X - a)) is irreducible over
 \texttt{K(a)} where \texttt{spad}(p(a) = 0).
\end{verbatim}

Implementation == add

\begin{verbatim}
 import PolynomialCategoryQuotientFunctions(IndexedExponents K,
 K, Z, SparseMultivariatePolynomial(Z, K), AN)

UPCF2 ==> UnivariatePolynomialCategoryFunctions2

fact : (UP, List K) -> FR
ifactor : (SUP, List K) -> Factored SUP
extend : (UP, Z) -> FR
allk : List AN -> List K
downpoly: UP -> UPQ
liftpoly: UPQ -> UP
irred? : UP -> Boolean
\end{verbatim}
allk l == removeDuplicates concat [kernels x for x in l]
liftpoly p == map(x +-> x::AN, p)$UPCF2(Q, UPQ, AN, UP)
downpoly p == map(x +-> retract(x)@Q, p)$UPCF2(AN, UP, Q, UPQ)
ifactor(p, l) == (fact(p pretend UP, l)) pretend Factored(SUP)
factor p == fact(p, allk coefficients p)

factor(p, l) ==
    fact(p, allk removeDuplicates concat(1, coefficients p))

split p ==
    fp := factor p
    unit(fp) *
    _*/[extend(fc.factor, fc.exponent) for fc in factors fp]

extend(p, n) ==
    -- one? degree p => primeFactor(p, n)
    (degree p = 1) => primeFactor(p, n)
    q := monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP
    primeFactor(q, n) * split((p exquo q)::UP) ** (n::N)

doublyTransitive? p ==
    irred? p and irred?((p exquo
        (monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP))::UP)

irred? p ==
    fp := factor p
    -- one? numberOfFactors fp and one? nthExponent(fp, 1)
    (numberOfFactors fp = 1) and (nthExponent(fp, 1) = 1)

fact(p, l) ==
    -- one? degree p => primeFactor(p, 1)
    (degree p = 1) => primeFactor(p, 1)
    empty? l =>
        dr := factor(downpoly p)$RationalFactorize(UPQ)
        (liftpoly unit dr) *
        _*/[primeFactor(liftpoly dc.factor, dc.exponent)
            for dc in factors dr]
        q := minPoly(alpha := "max"/l)$AN
        newl := remove((x:K):Boolean +-> alpha = x, l)
        sae := SimpleAlgebraicExtension(AN, SUP, q)
        ups := SparseUnivariatePolynomial sae
        fr := factor(map(x +-> reduce univariate(x, alpha, q), p)_
            $UPCF2(AN, UP, sae, ups),
            x +-> ifactor(x, newl))$InnerAlgFactor(AN, SUP, sae, ups)
        newalpha := alpha::AN
        map(((x:sae):AN +-> (lift(x)$sae) newalpha, unit fr)_
            $UPCF2(sae, ups, AN, UP) *
            _*/[primeFactor(map(((y:sae):AN +-> (lift(y)$sae) newalpha), fc.factor)_
                $UPCF2(sae, ups, AN, UP),
                newalpha, fc.factor)
PACKAGE INTPACK ANNA_NUMERIC_INTEGRATIONPACKAGE

fc.exp) for fc in factors fr]

-- ALGFACT.dotabb --

"ALGFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ALGFACT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"ALGFACT" -> "ACF"

-- AnnaNumericalIntegrationPackage.input --

)set break resume
)sys rm -f AnnaNumericalIntegrationPackage.output
)spool AnnaNumericalIntegrationPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show AnnaNumericalIntegrationPackage
--E 1

)spool
)lisp (bye)

-- AnnaNumericalIntegrationPackage.help --

====================================================================
AnnaNumericalIntegrationPackage examples
====================================================================

AnnaNumericalIntegrationPackage is a package of functions for the
category NumericalIntegrationCategory with measure and integrate.

See Also:
o )show AnnaNumericalIntegrationPackage
AnnaNumericalIntegrationPackage (INTPACK)

Exports:
   integrate  measure

— package INTPACK AnnaNumericalIntegrationPackage —

)abbrev package INTPACK AnnaNumericalIntegrationPackage
++ Author: Brian Dupee
++ Date Created: August 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{AnnaNumericalIntegrationPackage} is a \axiom{package}
++ of functions for the \axiom{category}
++ \axiomType{NumericalIntegrationCategory}
++ with \axiom{measure}, and \axiom{integrate}.

AnnaNumericalIntegrationPackage(): EE == II where

EDF ==> Expression DoubleFloat
DF ==> DoubleFloat
EF ==> Expression Float
F ==> Float
INT ==> Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
SBOCF ==> SegmentBinding OrderedCompletion Float
LSOCF ==> List Segment OrderedCompletion Float
SGCF ==> Segment OrderedCompletion Float
GCF ==> OrderedCompletion Float
LS ==> List Symbol
S ==> Symbol
LST ==> List String
ST ==> String
RT ==> RoutinesTable
NIA ==> Record(var:S, fn:EDF, range:SOCDF, abserr:DF, relerr:DF)
MDNIA ==> Record(fn:EDF, range:List SOCDF, abserr:DF, relerr:DF)
IFL ==> List(Record(ifail:Integer, instruction:String))
Entry ==> Record(chapter:String, type:String, domainName: String,

Measure ==> Record(measure:F, name:ST, explanations:LST, extra:Result)
EE => with

   integrate: (EF,SOCF,F,F,RT) -> Result
++ integrate(exp, a..b, epsrel, routines) is a top level ANNA function
++ to integrate an expression, {\tt exp}, over a given range {\tt a}
++ to {\tt b} to the required absolute and relative accuracy using
++ the routines available in the RoutinesTable provided.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory}
++ to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.

   integrate: NumericalIntegrationProblem -> Result
++ integrate(IntegrationProblem) is a top level ANNA function
++ to integrate an expression over a given range or ranges
++ to the required absolute and relative accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.

   integrate: (EF,SOCF,F,F) -> Result
++ integrate(exp, a..b, epsabs, epsrel) is a top level ANNA function
++ to integrate an expression, {\tt exp}, over a given range {\tt a}
++ to {\tt b} to the required absolute and relative accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.

\textbf{integrate: (EF,SDCF,F) -> Result}
++ integrate(exp, a..b, epsrel) is a top level ANNA
++ function to integrate an expression, \{tt exp\}, over a given
++ range \{tt a\} to \{tt b\} to the required relative accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ If epsrel = 0, a default absolute accuracy is used.

\textbf{integrate: (EF,SDCF) -> Result}
++ integrate(exp, a..b) is a top
++ level ANNA function to integrate an expression, \{tt exp\},
++ over a given range \{tt a\} to \{tt b\}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.

\textbf{integrate:(EF,LSOCF) -> Result}
++ integrate(exp, [a..b,c..d,...]) is a top
++ level ANNA function to integrate a multivariate expression, \{tt exp\},
++ over a given set of ranges.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.
integrate:(EF,LSOCF,F) -> Result
++ integrate(exp, [a..b,c..d,...], epsrel) is a top
++ level ANNA function to integrate a multivariate expression, \{\tt exp\},
++ over a given set of ranges to the required relative
++ accuracy.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{measure}.
++
++ It then performs the integration of the given expression
++ on that \texttt{domain}.
++
++ If epsrel = 0, a default absolute accuracy is used.

integrate:(EF,LSOCF,F,F) -> Result
++ integrate(exp, [a..b,c..d,...], epsabs, epsrel) is a top
++ level ANNA function to integrate a multivariate expression, \{\tt exp\},
++ over a given set of ranges to the required absolute and relative
++ accuracy.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{measure}.
++
++ It then performs the integration of the given expression
++ on that \texttt{domain}.

integrate:(EF,LSOCF,F,F,RT) -> Result
++ integrate(exp, [a..b,c..d,...], epsabs, epsrel, routines) is a top
++ level ANNA function to integrate a multivariate expression, \{\tt exp\},
++ over a given set of ranges to the required absolute and relative
++ accuracy, using the routines available in the RoutinesTable provided.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{measure}.
++
++ It then performs the integration of the given expression
++ on that \texttt{domain}.

measure:NumericalIntegrationProblem -> Measure
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine for solving the numerical integration
++ problem defined by \axiom{prob}.
++ It calls each \axiom{domain} of \axiom{category}
++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
++ and returns the best
++ i.e. the name of the most appropriate domain and any other relevant
++ information.

measure:(NumericalIntegrationProblem,RT) -> Measure
++ measure(prob,R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical integration
++ problem defined by \axiom{prob}.
++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
++ and returns the best
++ i.e. the name of the most appropriate domain and any other relevant
++ information.

integrate:(EF,SBOCF,ST) -> Union(Result,"failed")
++ integrate(exp, x = a..b, "numerical") is a top level ANNA function to
++ integrate an expression, \{tt exp\}, over a given range, \{tt a\}
++ to \{tt b\}.
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++ Default values for the absolute and relative error are used.
++ It is an error if the last argument is not \{tt "numerical"\}.

integrate:(EF,SBOCF,S) -> Union(Result,"failed")
++ integrate(exp, x = a..b, numerical) is a top level ANNA function to
++ integrate an expression, \{tt exp\}, over a given range, \{tt a\}
++ to \{tt b\}.
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++ Default values for the absolute and relative error are used.
++ It is an error if the last argument is not \tt numerical.

II ==> add

zeroMeasure: Measure -> Result
scriptedVariables?: MDNIA -> Boolean
preAnalysis: (Union(nia:NIA,mdnia:MDNIA),RT) -> RT
measureSpecific: (ST,RT,Union(nia:NIA,mdnia:MDNIA)) -> Record(measure:F,explanations:LST,extra:Result)
changeName: (Result,ST) -> Result
recoverAfterFail: (Union(nia:NIA,mdnia:MDNIA),RT,Measure,INT,Result) -> Record(a:Result,b:Measure)
better?: (Result,Result) -> Boolean
integrateConstant: (EF,SOCF) -> Result
integrateConstantList: (EF,LSOCF) -> Result
integrateArgs: (NumericalIntegrationProblem,RT) -> Result
integrateSpecific: (Union(nia:NIA,mdnia:MDNIA),ST,Result) -> Result

import ExpertSystemToolsPackage

integrateConstantList(exp:EF,ras:LSOCF):Result ==
c:OCF := ((retract(exp)@F)$EF)::OCF
b := [hi(j)-lo(j) for j in ras]
c := c*reduce((x,y) +-> x*y,b)
a := coerce(c)$AnyFunctions1(OCF)
text := coerce("Constant Function")$AnyFunctions1(ST)
construct([[result@S,a],[method@S,text]])$Result

integrateConstant(exp:EF,ra:SOCF):Result ==
c := (retract(exp)@F)$EF
r:OCF := (c::OCF)*(hi(ra)-lo(ra))
a := coerce(r)$AnyFunctions1(OCF)
text := coerce("Constant Function")$AnyFunctions1(ST)
construct([[result@Symbol,a],[method@Symbol,text]])$Result

zeroMeasure(m:Measure):Result ==
a := coerce(0$DF)$AnyFunctions1(DF)
text := coerce("Constant Function")$AnyFunctions1(String)
r := construct([[[result@Symbol,a],[method@Symbol,text]]])$Result
concat(measure2Result m,r)$ExpertSystemToolsPackage

scriptedVariables?(mdnia:MDNIA):Boolean ==
vars:List Symbol := variables(mdnia.fn)$EDF
var1 := first(vars)$(List Symbol)
not scripted?(var1) => false
name1 := name(var1)$Symbol
for i in 2..# vars repeat
not ((scripted?(vars.i)$Symbol) and (name1 = name(vars.i)$Symbol)) =>
  return false
true


```plaintext
preAnalysis(args: Union(nia: NIA, mdnia: MDNIA), t: RT): RT ==
  import RT
  r: RT := selectIntegrationRoutines t
  args case nia =>
    arg: NIA := args.nia
    rangeIsFinite(arg) $d01AgentsPackage case finite =>
      selectFiniteRoutines r
      selectNonFiniteRoutines r
      selectMultiDimensionalRoutines r
  changeName(ans: Result, name: ST): Result ==
    sy: S := coerce(name "Answer") $S
    anyAns: Any := coerce(ans) $AnyFunctions1(Result)
    construct([sy, anyAns]) $Result
  measureSpecific(name: ST, R: RT, args: Union(nia: NIA, mdnia: MDNIA)):
    Record(measure: F, explanations: ST, extra: Result) ==
      args case nia =>
        arg: NIA := args.nia
        name = "d01ajfAnnaType" => measure(R, arg) $d01ajfAnnaType
        name = "d01akfAnnaType" => measure(R, arg) $d01akfAnnaType
        name = "d01alfAnnaType" => measure(R, arg) $d01alfAnnaType
        name = "d01amfAnnaType" => measure(R, arg) $d01amfAnnaType
        name = "d01anfAnnaType" => measure(R, arg) $d01anfAnnaType
        name = "d01apfAnnaType" => measure(R, arg) $d01apfAnnaType
        name = "d01aqfAnnaType" => measure(R, arg) $d01aqfAnnaType
        name = "d01asfAnnaType" => measure(R, arg) $d01asfAnnaType
        name = "d01TransformFunctionType" =>
          measure(R, arg) $d01TransformFunctionType
        error("measureSpecific", "invalid type name: " name) $ErrorFunctions
      args case mdnia =>
        arg2: MDNIA := args.mdnia
        name = "d01gbfAnnaType" => measure(R, arg2) $d01gbfAnnaType
        name = "d01fcfAnnaType" => measure(R, arg2) $d01fcfAnnaType
        error("measureSpecific", "invalid type name: " name) $ErrorFunctions
        error("measureSpecific", "invalid type name") $ErrorFunctions
  measure(a: NumericalIntegrationProblem, R: RT): Measure ==
    args: Union(nia: NIA, mdnia: MDNIA) := retract(a) $NumericalIntegrationProblem
   sofar := 0$F
    best := "none" :: ST
    routs := copy R
    routs := preAnalysis(args, routs)
    empty?(routs) $RT =>
      error("measure", "no routines found") $ErrorFunctions
    rout := inspect(routs) $RT
    e := retract(rout.entry) $AnyFunctions1(Entry)
    meth: LST := ["Trying " e.type " integration routines"]
    ext := empty() $Result
```
for i in 1..# routs repeat
  rout := extract!(routs)$RT
  e := retract(rout.entry)$AnyFunctions1(Entry)
  n := e.domainName
  if e.defaultMin > sofar then
    m := measureSpecific(n,R,args)
    if m.measure > sofar then
      sofar := m.measure
      best := n
      ext := concat(m.extra,ext)$ExpertSystemToolsPackage
  str:LST := [string(rout.key)$S "measure: " outputMeasure(m.measure)
               " - " m.explanations]
else
  str:LST := [string(rout.key)$S " is no better than other routines"]
meth := append(meth,str)$LST
[sofar,best,meth,ext]

measure(a: NumericalIntegrationProblem): Measure ==
measure(a,routines()$RT)

integrateSpecific(args: Union(nia:NIA, mdnia: MDNIA), n: ST, ex: Result): Result ==
  args case nia =>
    arg:NIA := args.nia
    n = "d01ajfAnnaType" => numericalIntegration(arg, ex)$d01ajfAnnaType
    n = "d01TransformFunctionType" =>
      numericalIntegration(arg, ex)$d01TransformFunctionType
    n = "d01amfAnnaType" => numericalIntegration(arg, ex)$d01amfAnnaType
    n = "d01afpAnnaType" => numericalIntegration(arg, ex)$d01afpAnnaType
    n = "d01aqfAnnaType" => numericalIntegration(arg, ex)$d01aqfAnnaType
    n = "d01alfAnnaType" => numericalIntegration(arg, ex)$d01alfAnnaType
    n = "d01akfAnnaType" => numericalIntegration(arg, ex)$d01akfAnnaType
    n = "d01amfAnnaType" => numericalIntegration(arg, ex)$d01amfAnnaType
    n = "d01asfAnnaType" => numericalIntegration(arg, ex)$d01asfAnnaType
    error("integrateSpecific", "invalid type name: " n)$ErrorFunctions
  args case mdnia =>
    arg2: MDNIA := args.mdnia
    n = "d01gbfAnnaType" => numericalIntegration(arg2, ex)$d01gbfAnnaType
    n = "d01fcfAnnaType" => numericalIntegration(arg2, ex)$d01fcfAnnaType
    error("integrateSpecific", "invalid type name: " n)$ErrorFunctions
error("integrateSpecific", "invalid type name: " n)$ErrorFunctions

better?(r: Result, s: Result): Boolean ==
  a1 := search("abserr":: S, r)$Result
  a1 case "failed" => false
  abserr1 := retract(a1)$AnyFunctions1(DF)
  negative?(abserr1) => false
  a2 := search("abserr":: S, s)$Result
  a2 case "failed" => true
  abserr2 := retract(a2)$AnyFunctions1(DF)
  negative?(abserr2) => true
(abserr1 < abserr2) -- true if r.abserr better than s.abserr

recoverAfterFail(n:Union(nia:NIA,mdnia:MDNIA),routs:RT,m:Measure,iint:INT, r:Result):Record(a:Result,b:Measure) ==

bestName := m.name
while positive?(iint) repeat
  routineName := m.name
  s := recoverAfterFail(routs,routineName(1..6),iint)$RoutinesTable
  s case "failed" => iint := 0
  if s = "changeEps" then
    nn := n.nia
    zero?(nn.abserr) =>
      nn.abserr := 1.0e-8 :: DF
    m := measure(n::NumericalIntegrationProblem,routs)
    zero?(m.measure) => iint := 0
    r := integrateSpecific(n,m.name,m.extra)
    iint := 0
  rn := routineName(1..6)
  buttVal := getButtonValue(rn,"functionEvaluations")$AttributeButtons
  if (s = "incrFunEvals") and (buttVal < 0.8) then
    increase(rn,"functionEvaluations")$AttributeButtons
  if s = "increase tolerance" then
    (n.nia).relerr := (n.nia).relerr*(10.0::DF)
  if s = "decrease tolerance" then
    (n.nia).relerr := (n.nia).relerr/(10.0::DF)
  fl := coerce(s)$AnyFunctions1(ST)
  flrec:Record(key:S,entry:Any):=[failure@S,fl]
  m2 := measure(n::NumericalIntegrationProblem,routs)
  zero?(m2.measure) => iint := 0
  r2:Result := integrateSpecific(n,m2.name,m2.extra)
  better?(r,r2) =>
    m.name := m2.name
    insert!(flrec,r)$Result
    bestName := m2.name
    m := m2
    insert!(flrec,r2)$Result
  r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
  iany := search(ifail@S,r2)$Result
  iany case "failed" => iint := 0
  iint := retract(iany)$AnyFunctions1(INT)
  m.name := bestName
  [r,m]

integrateArgs(prob:NumericalIntegrationProblem,t:RT):Result ==
  args:Union(nia:NIA,mdnia:MDNIA) := retract(prob)$NumericalIntegrationProblem
  routs := copy(t)$RT
  if args case mdnia then
    arg := args.mdnia
    v := (# variables(arg.fn))
  not scriptedVariables?(arg) =>
error("MultiDimensionalNumericalIntegrationPackage",
"invalid variable names")$ErrorFunctions
(v ~= # arg.range)@Boolean =>
error("MultiDimensionalNumericalIntegrationPackage",
"number of variables do not match number of ranges")$ErrorFunctions
m := measure(prob,routs)
zero?(m.measure) => zeroMeasure m
r := integrateSpecific(args,m.name,m.extra)
iany := search(ifail$S,r)$Result
iint := 0$INT
if (iany case Any) then
  iint := retract(iany)$AnyFunctions1(INT)
if positive?(iint) then
tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
r := tu.a
m := tu.b
r := concat(measure2Result m,r)$ExpertSystemToolsPackage
n := m.name
nn:ST :=
(# n > 14) => "d01transform"
n(1..6)
expl := getExplanations(routs,nn)$RoutinesTable
expla := coerce(expl)$AnyFunctions1(LST)
explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
r := concat(construct([explaa]),r)
args case nia =>
  att := showAttributes(args.nia)$IntegrationFunctionsTable
  att case "failed" => r
  concat(att2Result att,r)$ExpertSystemToolsPackage
r
integrate(args:NumericalIntegrationProblem):Result ==
  integrateArgs(args,routines()$RT)
  Var:LS := variables(exp)$EF
  empty?(Var)$LS => integrateConstant(exp,ra)
  args:NIA := [first(Var)$LS,ef2edf exp,socf2socdf ra,f2df epsabs,f2df epsrel]
  integrateArgs(args::NumericalIntegrationProblem,r)
  integrate(exp,ra,epsabs,epsrel,routines()$RT)
integrate(exp:EF,ra:SOCF,err:F):Result ==
  positive?(err)$F => integrate(exp,ra,0$F,err)
  integrate(exp,ra,1.0E-5,err)
integrate(exp:EF,ra:SOCF):Result == integrate(exp,ra,0$F,1.0E-5)
integrate(exp:EF,sb:SBOCF, st:ST) ==
st = "numerical" => integrate(exp,segment sb)
"failed"

integrate(exp:EF,sb:SBOCF,s:S) ==
s = (numerical::Symbol) => integrate(exp,segment sb)
"failed"
vars := variables(exp)$EF
empty?(vars)$LS => integrateConstantList(exp,ra)
args:MDNIA := [ef2edf exp,convert ra,f2df epsabs,f2df epsrel]
integrateArgs(args::NumericalIntegrationProblem,r)

integrate(exp,ra,epsabs,epsrel,routines()$RT)

integrate(exp:EF,ra:LSOCF,epsrel:F):Result ==
zero? epsrel => integrate(exp,ra,1.0e-6,epsrel)
integrate(exp,ra,0$F,epsrel)

integrate(exp:EF,ra:LSOCF):Result == integrate(exp,ra,1.0e-4)

---

— INTPACK.dotabb —

"INTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPACK"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"INTPACK" -> "TBAGG"

---

package OPTPACK AnnaNumericalOptimizationPackage

— AnnaNumericalOptimizationPackage.input —

)set break resume
)sys rm -f AnnaNumericalOptimizationPackage.output
)spool AnnaNumericalOptimizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
AnnaNumericalOptimizationPackage examples

AnnaNumericalOptimizationPackage is a package of functions for the NumericalOptimizationCategory with measure and optimize.

See Also:
o show AnnaNumericalOptimizationPackage

Exports:
goodnessOfFit measure optimize

package OPTPACK AnnaNumericalOptimizationPackage —
+++ Date Last Updated: December 1997
+++ Description:
+++ \axiomType{AnnaNumericalOptimizationPackage} is a \axiom{package} of
+++ functions for the \axiomType{NumericalOptimizationCategory}
+++ with \axiom{measure} and \axiom{optimize}.

\textbf{AnnaNumericalOptimizationPackage}(): EE == II where

\begin{verbatim}
EDF ==> Expression DoubleFloat
LEDF ==> List Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
LCDF ==> List OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
LDFC ==> List OrderedCompletion Float
OCF ==> OrderedCompletion Float
LEF ==> List Expression Float
EF ==> Expression Float
LF ==> List Float
F ==> Float
LS ==> List Symbol
LST ==> List String
INT ==> Integer
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
IFL ==> List(Record(ifail:Integer, instruction:String))
Entry ==> Record(chapter:String, type:String, domainName: String,
Measure ==> Record(measure:F, name:String, explanations:List String)
Measure2 ==> Record(measure:F, explanations:String)
RT ==> RoutinesTable
UNOALSA ==> Union(noa:NOA, lsa:LSA)

EE ==> with

measure: NumericalOptimizationProblem -> Measure
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical optimization problem defined by
++ \axiom{prob} by checking various attributes of the functions and
++ calculating a measure of compatibility of each routine to these
++ attributes.
++
++ It calls each \axiom{domain} of \axiom{category}
++ \axiomType{NumericalOptimizationCategory} in turn to calculate all
++ measures and returns the best i.e. the name of the most
++ appropriate domain and any other relevant information.

measure: (NumericalOptimizationProblem, RT) -> Measure
++ measure(prob, R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical optimization problem defined by
++ \texttt{prob} by checking various attributes of the functions and
++ calculating a measure of compatibility of each routine to these
++ attributes.
++
++ It calls each \texttt{domain} listed in \texttt{Routines} of \texttt{category}
++ \texttt{NumericalOptimizationCategory} in turn to calculate all
++ measures and returns the best i.e. the name of the most
++ appropriate domain and any other relevant information.

optimize:(NumericalOptimizationProblem,RT) -> Result
++ optimize(prob,routines) is a top level ANNA function to
++ minimize a function or a set of functions with any constraints
++ as defined within \texttt{prob}.
++
++ It iterates over the \texttt{domains} listed in \texttt{routines} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

optimize:NumericalOptimizationProblem -> Result
++ optimize(prob) is a top level ANNA function to
++ minimize a function or a set of functions with any constraints
++ as defined within \texttt{prob}.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

goodnessOfFit:NumericalOptimizationProblem -> Result
++ goodnessOfFit(prob) is a top level ANNA function to
++ check to goodness of fit of a least squares model
++ as defined within \texttt{prob}.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.
++ It then calls the numerical routine \texttt{E04YCF} to get estimates
++ of the variance-covariance matrix of the regression coefficients of
++ the least-squares problem.
++
++ It thus returns both the results of the optimization and the
++ variance-covariance calculation.

optimize:(EF,LF,LOCF,LEF,LOCF) -> Result
++ optimize(f,start,lower,cons,upper) is a top level ANNA function to
++ minimize a function, \texttt{f}, of one or more variables with the
++ given constraints.
++ These constraints may be simple constraints on the variables
++ in which case \texttt{cons} would be an empty list and the bounds on
++ those variables defined in \texttt{lower} and \texttt{upper}, or a
++ mixture of simple, linear and non-linear constraints, where
++ \texttt{cons} contains the linear and non-linear constraints and
++ the bounds on these are added to \texttt{upper} and \texttt{lower}.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{optimize:}\langle\texttt{EF,LF,LOCF,LOCF}\rangle \to \texttt{Result}
++ \texttt{optimize(f,start,lower,upper)} is a top level ANNA function to
++ minimize a function, \texttt{f}, of one or more variables with
++ simple constraints. The bounds on
++ the variables are defined in \texttt{lower} and \texttt{upper}.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{optimize:}\langle\texttt{EF,LF}\rangle \to \texttt{Result}
++ \texttt{optimize(f,start)} is a top level ANNA function to
++ minimize a function, \texttt{f}, of one or more variables without
++ constraints.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{optimize:}\langle\texttt{LEF,LF}\rangle \to \texttt{Result}
++ \texttt{optimize(lf,start)} is a top level ANNA function to
++ minimize a set of functions, \texttt{lf}, of one or more variables
++ without constraints i.e. a least-squares problem.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.

goodnessOfFit: (LEF, LF) -> Result
++ goodnessOfFit(lf, start) is a top level ANNA function to
++ check to goodness of fit of a least squares model i.e. the minimization
++ of a set of functions, \axiom{lf}, of one or more variables without
++ constraints.
++
++ The parameter \axiom{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.
++ It then calls the numerical routine \axiomType{E04YCF} to get estimates
++ of the variance-covariance matrix of the regression coefficients of
++ the least-squares problem.
++
++ It thus returns both the results of the optimization and the
++ variance-covariance calculation.

++ goodnessOfFit(lf, start) is a top level function to iterate over
++ the \axiom{domains} of \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.
++ It then checks the goodness of fit of the least squares model.

II ==> add

preAnalysis: RT -> RT
zeroMeasure: Measure -> Result
optimizeSpecific: (UNOALSA, String) -> Result
measureSpecific: (String, RT, UNOALSA) -> Measure2
changeName: (Result, String) -> Result
recoverAfterFail: (UNOALSA, RT, Measure, INT, Result) -> Record(a: Result, b: Measure)
constant: UNOALSA -> Union(DF, "failed")
optimizeConstant: DF -> Result

import ExpertSystemToolsPackage, e04AgentsPackage, NumericalOptimizationProblem

constant(args: UNOALSA): Union(DF, "failed") ==
  args case noa =>
    Args := args.noa
    f := Args.fn
retractIfCan(f)@Union(DoubleFloat,"failed")
"failed"

optimizeConstant(c:DF): Result ==
a := coerce(c)$AnyFunctions1(DF)
text := coerce("Constant Function")$AnyFunctions1(String)
construct([[[objf@Symbol,a],[method@Symbol,text]]])$Result

preAnalysis(args:UNOALSA,t:RT):RT ==
r := selectOptimizationRoutines(t)$RT
args case lsa =>
selectSumOfSquaresRoutines(r)$RT
r

zeroMeasure(m:Measure):Result ==
a := coerce(0$F)$AnyFunctions1(F)
text := coerce("Zero Measure")$AnyFunctions1(String)
r := construct([[[objf@Symbol,a],[method@Symbol,text]]])$Result
concat(measure2Result m,r)

measureSpecific(name:String,R:RT,args:UNOALSA): Measure2 ==
args case noa =>
arg:NOA := args.noa
name = "e04dgfAnnaType" => measure(R,arg)$e04dgfAnnaType
name = "e04fdfAnnaType" => measure(R,arg)$e04fdfAnnaType
name = "e04gcgAnnaType" => measure(R,arg)$e04gcgAnnaType
name = "e04jafAnnaType" => measure(R,arg)$e04jafAnnaType
name = "e04mbfAnnaType" => measure(R,arg)$e04mbfAnnaType
name = "e04nafAnnaType" => measure(R,arg)$e04nafAnnaType
name = "e04ucfAnnaType" => measure(R,arg)$e04ucfAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions
args case lsa =>
arg2:LSA := args.lsa
name = "e04fdfAnnaType" => measure(R,arg2)$e04fdfAnnaType
name = "e04gcgAnnaType" => measure(R,arg2)$e04gcgAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions
error("measureSpecific","invalid argument type")$ErrorFunctions

measure(Args:NumericalOptimizationProblem,R:RT):Measure ==
args:UNOALSA := retract(Args)$NumericalOptimizationProblem
sofar := 0$F
best := "none" :: String
routs := copy R
routs := preAnalysis(args,routs)
empty?(routs)$RT =>
error("measure", "no routines found")$ErrorFunctions
rout := inspect(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
meth := empty()$(List String)
for i in 1..# routs repeat
rout := extract!(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
n := e.domainName
if e.defaultMin > sofar then
m := measureSpecific(n,R,args)
if m.measure > sofar then
sofar := m.measure
best := n
str := [concat(concat([string(rout.key)$Symbol,"measure: ",
outputMeasure(m.measure)," - "],
m.explanations)$((List String))$String]
else
str := [concat([string(rout.key)$Symbol," is no better than other routines"])$String]
meth := append(meth,str)$((List String))
[sofar,best,meth]

measure(args:NumericalOptimizationProblem):Measure == measure(args,routines()$RT)
optimizeSpecific(args:UNOALSA,name:String):Result ==
args case noa =>
arg:NOA := args.noa
name = "e04dgfAnnaType" => numericalOptimization(arg)$e04dgfAnnaType
name = "e04fdfAnnaType" => numericalOptimization(arg)$e04fdfAnnaType
name = "e04gcfAnnaType" => numericalOptimization(arg)$e04gcfAnnaType
name = "e04jafAnnaType" => numericalOptimization(arg)$e04jafAnnaType
name = "e04mbfAnnaType" => numericalOptimization(arg)$e04mbfAnnaType
name = "e04nafAnnaType" => numericalOptimization(arg)$e04nafAnnaType
name = "e04ucfAnnaType" => numericalOptimization(arg)$e04ucfAnnaType
error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
args case lsa =>
arg:LSA := args.lsa
name = "e04fdfAnnaType" => numericalOptimization(arg2)$e04fdfAnnaType
name = "e04gcfAnnaType" => numericalOptimization(arg2)$e04gcfAnnaType
error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
error("optimizeSpecific","invalid type name: " name)$ErrorFunctions

changeName(ans:Result,name:String):Result ==
st:String := concat([name,"Answer"])$String
sy:Symbol := coerce(st)$Symbol
anyAns:Any := coerce(ans)$AnyFunctions1(Result)
construct([[sy,anyAns]])$Result

recoverAfterFail(args:UNOALSA,routs:RT,m:Measure,
iint:INT,r:Result):Record(a:Result,b:Measure) ==
while positive?(iint) repeat
routineName := m.name
s := recoverAfterFail(routs,routineName(1..6),iint)$RT
s case "failed" => iint := 0
(s = "no action")@Boolean => iint := 0
f1 := coerce(s)$AnyFunctions1(String)
f1rec:Record(key:Symbol,entry:Any):=[failure@Symbol,f1]
m2 := measure(args::NumericalOptimizationProblem,routs)
zero?(m2.measure) => iint := 0
r2:Result := optimizeSpecific(args,m2.name)
m := m2
insert!(f1rec,r2)$Result
r := concat(r2,changeName(r,routineName))
iany := search(ifail@Symbol,r2)$Result
iany case "failed" => iint := 0
iint := retract(iany)$AnyFunctions1(INT)
[r,m]

optimize(Args:NumericalOptimizationProblem,t:RT):Result ==
args:UNOALSA := retract(Args)$NumericalOptimizationProblem
routs := copy(t)$RT
c:Union(DF,"failed") := constant(args)
c case DF => optimizeConstant(c)
m := measure(Args,routs)
zero?(m.measure) => zeroMeasure m
r := optimizeSpecific(args,n := m.name)
iany := search(ifail@Symbol,r)$Result
iint := 0$INT
if (iany case Any) then
  iint := retract(iany)$AnyFunctions1(INT)
if positive?(iint) then
  tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
r := tu.a
m := tu.b
r := concat(measure2Result m,r)
expl := getExplanations(routs,n(1..6))$RoutinesTable
expla := coerce(expl)$AnyFunctions1(LST)
expla:a:Record(key:Symbol,entry:Any) := ["explanations":Symbol,expla]
r := concat(construct([expla]),r)
att:List String := optAttributes(args)
atta := coerce(att)$AnyFunctions1(List String)
attr:Record(key:Symbol,entry:Any) := [attributes@Symbol,atta]
insert!(attr,r)$Result

optimize(args:NumericalOptimizationProblem):Result ==
  optimize(args,routines()$RT)

goodnessOfFit(Args:NumericalOptimizationProblem):Result ==
  r := optimize(Args)
args1:UNOALSA := retract(Args)$NumericalOptimizationProblem
args1 case noa => error("goodnessOfFit","Not an appropriate problem")
args:LSA := args1.lsa
lf := args.lfn
n:INT := #(variables(args))
m:INT := # lf
me := search(method,r)$Result
me case "failed" => r
meth := retract(me)$AnyFunctions1(Result)
na := search(nameOfRoutine,meth)$Result
na case "failed" => r
name := retract(na)$AnyFunctions1(String)
temp:INT := (n*(n-1)) quo 2
ns:INT :=
    name = "e04fdAnnaType" => 6*n+(2+n)*m+1+max(1,temp)
    8*n+(n+2)*m+temp+1+max(1,temp)
    ns :=
ww := search(w,r)$Result
ww case "failed" => r
ws:MDF := retract(ww)$AnyFunctions1(MDF)
fr := search(objf,r)$Result
fr case "failed" => r
f := retract(fr)$AnyFunctions1(DF)
s := subMatrix(ws,1,1,ns,nv-1)$MDF
v := subMatrix(ws,1,1,nv,nv+n*n-1)$MDF
r2 := e04ycf(0,m,n,f,s,n,v,-1)$NagOptimisationPackage
concat(r,r2)

    args:NOA := 
        [ef2edf(f),f2df i for i in start],[ocf2ocdf j for j in lower],
        [ef2edf k for k in cons], [ocf2ocdf l for l in upper]
    optimize(args::NumericalOptimizationProblem)

optimize(f:EF,start:LF,lower:LOCF,upper:LOCF):Result ==
    optimize(f,start,lower,empty()$LEF,upper)

optimize(f:EF,start:LF):Result ==
    optimize(f,start,empty()$LOCF,empty()$LOCF)

optimize(lf:LEF,start:LF):Result ==
    args:LSA := [ef2edf i for i in lf],[f2df j for j in start]
    optimize(args::NumericalOptimizationProblem)

goodnessOfFit(lf:LEF,start:LF):Result ==
    args:LSA := [ef2edf i for i in lf],[f2df j for j in start]
    goodnessOfFit(args::NumericalOptimizationProblem)

—— OPTPACK.dotabb ——

"OPTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OPTPACK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"OPTPACK" -> "ALIST"
package ODEPACK AnnaOrdinaryDifferentialEquation-Package

-- AnnaOrdinaryDifferentialEquationPackage.input --

)set break resume
)sys rm -f AnnaOrdinaryDifferentialEquationPackage.output
)spool AnnaOrdinaryDifferentialEquationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AnnaOrdinaryDifferentialEquationPackage
--E 1

)spool
)lisp (bye)

-- AnnaOrdinaryDifferentialEquationPackage.help --

====================================================================
AnnaOrdinaryDifferentialEquationPackage examples
====================================================================

AnnaOrdinaryDifferentialEquationPackage is a package of functions for the
category OrdinaryDifferentialEquationsSolverCategory with measure and
solve.

See Also:
  o )show AnnaOrdinaryDifferentialEquationPackage
AnnaOrdinaryDifferentialEquationPackage (ODEPACK)

Exports:
measure solve

— package ODEPACK AnnaOrdinaryDifferentialEquationPackage —

)abbrev package ODEPACK AnnaOrdinaryDifferentialEquationPackage
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: December 1997
++ Description:
++ \texttt{AnnaOrdinaryDifferentialEquationPackage} is a \texttt{package}
++ of functions for the \texttt{category}
++ \texttt{OrdinaryDifferentialEquationsSolverCategory}
++ with \texttt{measure}, and \texttt{solve}.

AnnaOrdinaryDifferentialEquationPackage(): EE == II where

EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
LF ==> List Float
F ==> Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
MF ==> MachineFloat
LS ==> List Symbol
ST ==> String
CHAPTER 2. CHAPTER A

LST ==> List String
INT ==> Integer
RT ==> RoutinesTable
ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,_
g:EDF,abserr:DF,relerr:DF)
IFL ==> List(Record(ifail:Integer,instruction:String))
Entry ==> Record(chapter:String, type:String, domainName: String,
Measure ==> Record(measure:F,name:String, explanations:List String)

EE ==> with
  solve:(NumericalODEProblem) -> Result
  ++ solve(odeProblem) is a top level ANNA function to solve numerically a
  ++ system of ordinary differential equations i.e. equations for the
  ++ derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n], together
  ++ with starting values for x and y[1]..y[n] (called the initial
  ++ conditions), a final value of x, an accuracy requirement and any
  ++ intermediate points at which the result is required.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory}
  ++ to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library. The function predicts the likely most effective routine
  ++ by checking various attributes of the system of ODE's and calculating
  ++ a measure of compatibility of each routine to these attributes.
  ++
  ++ It then calls the resulting 'best' routine.

  solve:(NumericalODEProblem,RT) -> Result
  ++ solve(odeProblem,R) is a top level ANNA function to solve numerically a
  ++ system of ordinary differential equations i.e. equations for the
  ++ derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n], together
  ++ with starting values for x and y[1]..y[n] (called the initial
  ++ conditions), a final value of x, an accuracy requirement and any
  ++ intermediate points at which the result is required.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
  ++ the table of routines \axiom{R} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine 
++ by checking various attributes of the system of ODE’s and calculating 
++ a measure of compatibility of each routine to these attributes. 
++ It then calls the resulting ‘best’ routine.

solve:(VEF,F,F,LF) -> Result
++ solve(f,xStart,xEnd,yInitial) is a top level ANNA function to solve 
++ numerically a system of ordinary differential equations i.e. equations 
++ for the derivatives y[1]’..y[n]’ defined in terms of x,y[1]..y[n], 
++ together with a starting value for x and y[1]..y[n] (called the initial 
++ conditions) and a final value of x. A default value 
++ is used for the accuracy requirement. 
++ It iterates over the \texttt{domains} of 
++ \texttt{OrdinaryDifferentialEquationsSolverCategory} contained in 
++ the table of routines \texttt{R} to get the name and other 
++ relevant information of the the (domain of the) numerical 
++ routine likely to be the most appropriate, 
++ i.e. have the best \texttt{measure}. 
++ The method used to perform the numerical 
++ process will be one of the routines contained in the NAG numerical 
++ Library. The function predicts the likely most effective routine 
++ by checking various attributes of the system of ODE’s and calculating 
++ a measure of compatibility of each routine to these attributes. 
++ It then calls the resulting ‘best’ routine.

solve:(VEF,F,F,LF,EF,F) -> Result
++ solve(f,xStart,xEnd,yInitial,G,tol) is a top level ANNA function to solve 
++ numerically a system of ordinary differential equations, \texttt{f}, 
++ i.e. equations for the derivatives y[1]’..y[n]’ defined in terms 
++ of x,y[1]..y[n] from \texttt{xStart} to \texttt{xEnd} with the initial 
++ values for y[1]..y[n] (\texttt{yInitial}) to a tolerance \texttt{tol}. 
++ It iterates over the \texttt{domains} of 
++ \texttt{OrdinaryDifferentialEquationsSolverCategory} contained in 
++ the table of routines \texttt{R} to get the name and other 
++ relevant information of the the (domain of the) numerical 
++ routine likely to be the most appropriate, 
++ i.e. have the best \texttt{measure}. 
++ The method used to perform the numerical 
++ process will be one of the routines contained in the NAG numerical 
++ Library. The function predicts the likely most effective routine 
++ by checking various attributes of the system of ODE’s and calculating 
++ a measure of compatibility of each routine to these attributes. 
++ It then calls the resulting ‘best’ routine.

solve:(VEF,F,F,LF,EF,F) -> Result
++ solve(f,xStart,xEnd,yInitial,G,tol) is a top level ANNA function to
++ solve numerically a system of ordinary differential equations, 
++ \texttt{\texttt{axiom}(f)}, i.e. equations for the derivatives \texttt{y[i]}'..\texttt{y[n]}' defined in
++ terms of \texttt{x},\texttt{y[1]..y[n]} from \texttt{\texttt{axiom}(xStart)} to \texttt{\texttt{axiom}(xEnd)} with the
++ initial values for \texttt{y[i]..y[n]} (\texttt{\texttt{axiom}(yInitial)}) to a tolerance 
++ \texttt{\texttt{axiom}(tol)}. The calculation will stop if the function
++ \texttt{G(x,y[1],..,y[n])} evaluates to zero before \texttt{x = xEnd}.
++
++ It iterates over the \texttt{\texttt{axiom}(domains)} of 
++ \texttt{\texttt{axiomType}(OrdinaryDifferentialEquationsSolverCategory)} contained in
++ the table of routines \texttt{\texttt{axiom}(R)} to get the name and other 
++ relevant information of the the (domain of the) numerical 
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{\texttt{axiom}(measure)}. 
++
++ The method used to perform the numerical process will be one of the 
++ routines contained in the NAG numerical Library. The function 
++ predicts the likely most effective routine by checking various 
++ attributes of the system of ODE's and calculating a measure of 
++ compatibility of each routine to these attributes. 
++
++ It then calls the resulting 'best' routine.

\hspace{1cm}
solve: (\texttt{\texttt{VEF}},\texttt{F},\texttt{F},\texttt{LF},\texttt{LF},\texttt{F}) \rightarrow \texttt{Result}
++ \texttt{solve(f,xStart,xEnd,yInitial,intVals,tol)} is a top level ANNA function
++ to solve numerically a system of ordinary differential equations,
++ \texttt{\texttt{axiom}(f)}, i.e. equations for the derivatives \texttt{y[i]}'..\texttt{y[n]}' defined in
++ terms of \texttt{x},\texttt{y[1]..y[n]} from \texttt{\texttt{axiom}(xStart)} to \texttt{\texttt{axiom}(xEnd)} with the
++ initial values for \texttt{y[i]..y[n]} (\texttt{\texttt{axiom}(yInitial)}) to a tolerance 
++ \texttt{\texttt{axiom}(tol)}. The values of \texttt{y[1]..y[n]} will be output for the values 
++ of \texttt{x} in \texttt{\texttt{axiom}(intVals)}. 
++
++ It iterates over the \texttt{\texttt{axiom}(domains)} of 
++ \texttt{\texttt{axiomType}(OrdinaryDifferentialEquationsSolverCategory)} contained in
++ the table of routines \texttt{\texttt{axiom}(R)} to get the name and other 
++ relevant information of the the (domain of the) numerical 
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{\texttt{axiom}(measure)}. 
++
++ The method used to perform the numerical 
++ process will be one of the routines contained in the NAG numerical 
++ Library. The function predicts the likely most effective routine 
++ by checking various attributes of the system of ODE's and calculating 
++ a measure of compatibility of each routine to these attributes. 
++
++ It then calls the resulting 'best' routine.

\hspace{1cm}
solve: (\texttt{\texttt{VEF}},\texttt{F},\texttt{F},\texttt{LF},\texttt{EF},\texttt{LF},\texttt{F}) \rightarrow \texttt{Result}
++ \texttt{solve(f,xStart,xEnd,yInitial,G,intVals,tol)} is a top level ANNA function
++ to solve numerically a system of ordinary differential 
++ equations, \texttt{\texttt{axiom}(f)}, i.e. equations for the derivatives \texttt{y[i]}'..\texttt{y[n]}' 
++ defined in terms of \texttt{x},\texttt{y[1]..y[n]} from \texttt{\texttt{axiom}(xStart)} to \texttt{\texttt{axiom}(xEnd)}
++ with the initial values for \texttt{y[i]..y[n]} (\texttt{\texttt{axiom}(yInitial)}) to a
++
tolerance $\texttt{tol}$. The values of $y[1]..y[n]$ will be output for
the values of $x$ in $\texttt{intVals}$. The calculation will stop if the
function $G(x,y[1],..,y[n])$ evaluates to zero before $x = xEnd$.

It iterates over the $\texttt{domains}$ of
$\texttt{OrdinaryDifferentialEquationsSolverCategory}$ contained in
the table of routines $\texttt{R}$ to get the name and other
relevant information of the the (domain of the) numerical
routine likely to be the most appropriate,
 i.e. have the best $\texttt{measure}$.

The method used to perform the numerical
process will be one of the routines contained in the NAG numerical
Library. The function predicts the likely most effective routine
by checking various attributes of the system of ODE's and calculating
a measure of compatibility of each routine to these attributes.

It then calls the resulting 'best' routine.

solve : (\texttt{VEF}, \texttt{F}, \texttt{F}, \texttt{LF}, \texttt{EF}, \texttt{LF}, \texttt{F}, \texttt{F}) \to \texttt{Result}

solve($f, xStart, xEnd, yInitial, G, intVals, epsabs, epsrel$) is a top level
\texttt{ANNA} function to solve numerically a system of ordinary differential
equations, $\texttt{f}$, i.e.
equations for the derivatives $y[1]'..y[n]'$ defined in terms
of $x,y[1]..y[n]$ from $\texttt{xStart}$ to $\texttt{xEnd}$ with the initial
values for $y[1]..y[n]$ ($\texttt{yInitial}$) to an absolute error
requirement $\texttt{epsabs}$ and relative error $\texttt{epsrel}$.
The values of $y[1]..y[n]$ will be output for the values of $x$ in
$\texttt{intVals}$. The calculation will stop if the function
$G(x,y[1],..,y[n])$ evaluates to zero before $x = xEnd$.

It iterates over the $\texttt{domains}$ of
$\texttt{OrdinaryDifferentialEquationsSolverCategory}$ contained in
the table of routines $\texttt{R}$ to get the name and other
relevant information of the the (domain of the) numerical
routine likely to be the most appropriate,
i.e. have the best $\texttt{measure}$.

The method used to perform the numerical
process will be one of the routines contained in the NAG numerical
Library. The function predicts the likely most effective routine
by checking various attributes of the system of ODE's and calculating
a measure of compatibility of each routine to these attributes.

It then calls the resulting 'best' routine.

measure : (\texttt{NumericalODEProblem}) \to \texttt{Measure}

measure($\texttt{prob}$) is a top level \texttt{ANNA} function for identifying the most
appropriate numerical routine from those in the routines table
provided for solving the numerical ODE
problem defined by $\texttt{prob}$. 

++
It calls each `\text{domain}` of `\text{category}` in turn to calculate all measures and returns the best i.e. the name of the most appropriate domain and any other relevant information. It predicts the likely most effective NAG numerical Library routine to solve the input set of ODEs by checking various attributes of the system of ODEs and calculating a measure of compatibility of each routine to these attributes.

\text{measure}: (\text{NumericalODEProblem,RT}) \rightarrow \text{Measure}

It predicts the likely most effective NAG numerical Library routine to solve the input set of ODEs by checking various attributes of the system of ODEs and calculating a measure of compatibility of each routine to these attributes. It calls each `\text{domain}` listed in `\text{R}` of `\text{category}` in turn to calculate all measures and returns the best i.e. the name of the most appropriate domain and any other relevant information. It predicts the likely most effective NAG numerical Library routine to solve the input set of ODEs by checking various attributes of the system of ODEs and calculating a measure of compatibility of each routine to these attributes.

\text{II} \Rightarrow \text{add}

\text{import } \text{ODEA, Numerical}\text{ODEProblem}

\text{f2df: F} \rightarrow \text{DF}
\text{ef2edf: EF} \rightarrow \text{EDF}

\text{preAnalysis}: (\text{ODEA,RT}) \rightarrow \text{RT}

\text{zeroMeasure: Measure} \rightarrow \text{Result}

\text{measureSpecific}: (\text{ST,RT,ODEA}) \rightarrow \text{Record(measure:F, explanations:ST)}

\text{solveSpecific}: (\text{ODEA,ST}) \rightarrow \text{Result}

\text{changeName}: (\text{Result,ST}) \rightarrow \text{Result}

\text{recoverAfterFail}: (\text{ODEA,RT, Measure, Integer, Result}) \rightarrow \text{Record(a: Result, b: Measure)}

\text{f2df(f:F):DF} \rightarrow \text{(convert(f)@DF)$F}

\text{ef2edf(f:EF):EDF} \rightarrow \text{map(f2df,f)$ExpressionFunctions2(F,DF)}

\text{preAnalysis(args:ODEA,t:RT):RT} =
\text{rt} := \text{selectODEIVPRoutines(t)$RT}
\text{if positive?($\text{variables(args.g)}$) then}
\text{changeMeasure(rt,d02bbf@Symbol,getMeasure(rt,d02bbf@Symbol)*0.8)}
\text{if positive?($\text{args.intvals}$) then}
\text{changeMeasure(rt,d02bbf@Symbol,getMeasure(rt,d02bbf@Symbol)*0.8)}
\text{rt}

\text{zeroMeasure(m:Measure):Result} =
\text{a := coerce(0$F)$AnyFunctions1(F)
text := coerce("Zero Measure")$AnyFunctions1(ST)
r := construct([[result@Symbol,a],[method@Symbol,text]])$Result
concat(measure2Result m,r)$ExpertSystemToolsPackage

measureSpecific(name:ST,R:RT,ode:ODEA):Record(measure:F,explanations:ST) ==
  name = "d02bbfAnnaType" => measure(R,ode)$d02bbfAnnaType
  name = "d02bhfAnnaType" => measure(R,ode)$d02bhfAnnaType
  name = "d02cjfAnnaType" => measure(R,ode)$d02cjfAnnaType
  name = "d02ejfAnnaType" => measure(R,ode)$d02ejfAnnaType
  error("measureSpecific", "invalid type name: " name)$ErrorFunctions

  ode:ODEA := retract(ode)$NumericalODEProblem
  sofar := 0$F
  best := "none" :: ST
  routs := copy R
  routs := preAnalysis(ode,routs)
  empty?(routs)$RT =>
    error("measure", "no routines found")$ErrorFunctions
  rout := inspect(routs)$RT
  e := retract(rout.entry)$AnyFunctions1(Entry)
  meth := empty()$LST
  for i in 1..# routs repeat
    rout := extract!(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    n := e.domainName
    if e.defaultMin > sofar then
      m := measureSpecific(n,R,ode)
      if m.measure > sofar then
        sofar := m.measure
        best := n
        str:LST := [string(rout.key)$Symbol "measure: ",
                     outputMeasure(m.measure)$ExpertSystemToolsPackage " - ",
                     m.explanations]
      else
        str := [string(rout.key)$Symbol " is no better than other routines"]
    meth := append(meth,str)$LST
  [sofar,best,meth]

solveSpecific(ode:ODEA,n:ST):Result ==
  n = "d02bbfAnnaType" => ODESolve(ode)$d02bbfAnnaType
  n = "d02bhfAnnaType" => ODESolve(ode)$d02bhfAnnaType
  n = "d02cjfAnnaType" => ODESolve(ode)$d02cjfAnnaType
  n = "d02ejfAnnaType" => ODESolve(ode)$d02ejfAnnaType
  error("solveSpecific", "invalid type name: " n)$ErrorFunctions

changeName(ans:Result,name:ST):Result ==
  sy:Symbol := coerce(name "Answer")$Symbol
anyAns: Any := coerce(ans)$AnyFunctions1(Result)
construct([isy,anyAns])$Result

recoverAfterFail(ode: ODEA, routs: RT, m: Measure, iint: Integer, r: Result): Record(a: Result, b: Measure) ==
  while positive?(iint) repeat
    routineName := m.name
    s := recoverAfterFail(routs, routineName(1..6), iint)$RT
    s case "failed" => iint := 0
    if s = "increase tolerance" then
      ode.relerr := ode.relerr*(10.0::DF)
      ode.abserr := ode.abserr*(10.0::DF)
    if s = "decrease tolerance" then
      ode.relerr := ode.relerr/(10.0::DF)
      ode.abserr := ode.abserr/(10.0::DF)
    (s = "no action")@Boolean => iint := 0
    fl := coerce(s)$AnyFunctions1(ST)
    flrec: Record(key: Symbol, entry: Any) := [failure@Symbol, fl]
    m2 := measure(ode:: NumericalODEProblem, routs)
    zero?(m2.measure) => iint := 0
    r2: Result := solveSpecific(ode, m2.name)
    m := m2
    insert!(flrec, r2)$Result
    r := concat(r2, changeName(r, routineName))$ExpertSystemToolsPackage
    iany := search(ifail@Symbol, r)$Result
    iany case "failed" => iint := 0
    iint := retract(iany)$AnyFunctions1(Integer)
  [r, m]

solve(Ode: NumericalODEProblem, t: RT): Result ==
  ode: ODEA := retract(Ode)$NumericalODEProblem
  routs := copy(t)$RT
  m := measure(Ode, routs)
  zero?(m.measure) => zeroMeasure m
  r := solveSpecific(ode, n := m.name)
  iany := search(ifail@Symbol, r)$Result
  iint := 0$Integer
  if (iany case Any) then
    iint := retract(iany)$AnyFunctions1(Integer)
  if positive?(iint) then
    tu: Record(a: Result, b: Measure) := recoverAfterFail(ode, routs, m, iint, r)
    r := tu.a
    m := tu.b
    r := concat(measure2Result m, r)$ExpertSystemToolsPackage
    expla := getExplanations(routs, n(1..6))$RoutinesTable
    expla := coerce(expla)$AnyFunctions1(LST)
    explaa: Record(key: Symbol, entry: Any) := ["explanations": Symbol, expla]
    r := concat(construct([explaa]), r)
  iflist := showIntensityFunctions(ode)$ODEIntensityFunctionsTable
  iflist case "failed" => r
concat(iflist2Result iflist, r)$ExpertSystemToolsPackage

solve(ode:NumericalODEProblem):Result == solve(ode,routines()$RT)

d:ODEA := [f2df xStart,f2df xEnd,vector([ef2edf e for e in members f])$VEDF,
    [f2df i for i in yInitial], [f2df j for j in intVal],
    ef2edf G,f2df epsabs,f2df epsrel]
solve(d::NumericalODEProblem,routines()$RT)

solve(f,xStart,xEnd,yInitial,G,intVal,tol,tol)

solve(f,xStart,xEnd,yInitial,1$EF,intVal,tol)

solve(f,xStart,xEnd,y,G,empty()$LF,tol)

solve(f:VEF,xStart:F,xEnd:F,yInitial:LF,tol:F):Result ==
solve(f,xStart,xEnd,yInitial,1.0e-4)

— ODEPACK.dotabb —

"ODEPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPACK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ODEPACK" -> "ALIST"

package PDEPACK AnnaPartialDifferentialEquationPackage

— AnnaPartialDifferentialEquationPackage.input —
AnnaPartialDifferentialEquationPackage is an uncompleted package for the interface to NAG PDE routines. It has been realised that a new approach to solving PDEs will need to be created.

See Also:
- )show AnnaPartialDifferentialEquationPackage

AnnaPartialDifferentialEquationPackage (PDEPACK)

Exports:
- measure
- solve

--- package PDEPACK AnnaPartialDifferentialEquationPackage ---
++ Author: Brian Dupee
++ Date Created: June 1996
++ Date Last Updated: December 1997
++ Description:
++ AnnaPartialDifferentialEquationPackage is an uncompleted
++ package for the interface to NAG PDE routines. It has been realised that
++ a new approach to solving PDEs will need to be created.

AnnaPartialDifferentialEquationPackage(): EE == II where

LEDF ==> List Expression DoubleFloat
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
LEF ==> List Expression Float
EF ==> Expression Float
MEF ==> Matrix Expression Float
LF ==> List Float
F ==> Float
LS ==> List Symbol
ST ==> String
LST ==> List String
INT ==> Integer
NNI ==> NonNegativeInteger
RT ==> RoutinesTable
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
  dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
  f:List LEDF, st:ST, tol:DF)
IFL ==> List(Record(ifail:INT, instruction:ST))
Entry ==> Record(chapter:ST, type:ST, domainName: ST,
Measure ==> Record(measure:F, name:ST, explanations:LST)

EE ==> with
solve:(NumericalPDEProblem) -> Result
++ solve(PDEProblem) is a top level ANNA function to solve numerically
++ a system of partial differential equations.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of PDE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting ‘best’ routine.
++
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **
solve:(NumericalPDEProblem, RT) -> Result
   ++ solve(PDEProblem, routines) is a top level ANNA function to solve numerically a system
   ++ of partial differential equations.
   ++
   ++ The method used to perform the numerical
   ++ process will be one of the routines contained in the NAG numerical
   ++ Library. The function predicts the likely most effective routine
   ++ by checking various attributes of the system of PDE's and calculating
   ++ a measure of compatibility of each routine to these attributes.
   ++
   ++ It then calls the resulting 'best' routine.
   ++
   ++ ** At the moment, only Second Order Elliptic Partial Differential
   ++ Equations are solved **

solve:(F, F, F, NNI, NNI, LEF, List LEF, ST, DF) -> Result
   ++ solve(xmin, ymin, xmax, ymax, ngx, ngy, pde, bounds, st, tol) is a top level
   ++ ANNA function to solve numerically a system of partial differential
   ++ equations. This is defined as a list of coefficients \(\text{pde}\),
   ++ a grid \(\text{xmin}, \text{ymin}, \text{xmax}, \text{ymax}\), \(\text{ngx}, \text{ngy}\),
   ++ the boundary values \(\text{bounds}\) and a
   ++ tolerance requirement \(\text{tol}\). There is also a parameter
   ++ \(\text{st}\) which should contain the value "elliptic" if the PDE is
   ++ known to be elliptic, or "unknown" if it is uncertain. This causes the
   ++ routine to check whether the PDE is elliptic.
   ++
   ++ The method used to perform the numerical
   ++ process will be one of the routines contained in the NAG numerical
   ++ Library. The function predicts the likely most effective routine
   ++ by checking various attributes of the system of PDE's and calculating
   ++ a measure of compatibility of each routine to these attributes.
   ++
   ++ It then calls the resulting 'best' routine.
   ++
   ++ ** At the moment, only Second Order Elliptic Partial Differential
   ++ Equations are solved **

solve:(F, F, F, NNI, NNI, LEF, List LEF, ST) -> Result
   ++ solve(xmin, ymin, xmax, ymax, ngx, ngy, pde, bounds, st) is a top level
   ++ ANNA function to solve numerically a system of partial differential
   ++ equations. This is defined as a list of coefficients \(\text{pde}\),
   ++ a grid \(\text{xmin}, \text{ymin}, \text{xmax}, \text{ymax}\), \(\text{ngx}, \text{ngy}\),
   ++ the boundary values \(\text{bounds}\). A default value for tolerance is used. There is also a parameter
   ++ \(\text{st}\) which should contain the value "elliptic" if the PDE is
   ++ known to be elliptic, or "unknown" if it is uncertain. This causes the
   ++ routine to check whether the PDE is elliptic.
   ++
   ++ The method used to perform the numerical
   ++ process will be one of the routines contained in the NAG numerical
   ++ Library. The function predicts the likely most effective routine
   ++ by checking various attributes of the system of PDE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++ It then calls the resulting 'best' routine.
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **

measure:(NumericalPDEProblem) -> Measure
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical PDE
++ problem defined by \texttt{axiom(prob)}.
++ It calls each \texttt{\textbackslash axiom\{domain\}} of \texttt{\textbackslash axiom\{category\}}
++ \texttt{\textbackslash axiomType\{\textbackslash axiom\{PartialDifferentialEquationsSolverCategory\}\}} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical
++ Library routine to solve the input set of PDEs
++ by checking various attributes of the system of PDEs and calculating
++ a measure of compatibility of each routine to these attributes.

measure:(NumericalPDEProblem,RT) -> Measure
++ measure(prob,R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical PDE
++ problem defined by \texttt{axiom(prob)}.
++ It calls each \texttt{\textbackslash axiom\{domain\}} listed in \texttt{\textbackslash axiom\{R\}} of \texttt{\textbackslash axiom\{category\}}
++ \texttt{\textbackslash axiomType\{\textbackslash axiom\{PartialDifferentialEquationsSolverCategory\}\}} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical
++ Library routine to solve the input set of PDEs
++ by checking various attributes of the system of PDEs and calculating
++ a measure of compatibility of each routine to these attributes.

II ==> add

import PDEB, d03AgentsPackage, ExpertSystemToolsPackage, NumericalPDEProblem

zeroMeasure:Measure -> Result
measureSpecific:(ST,RT,PDEB) -> Record(measure:F,explanations:ST)
solveSpecific:(PDEB,ST) -> Result
changeName:(Result,ST) -> Result
recoverAfterFail:(PDEB,RT,Measure,Integer,Result) -> Record(a:Result,b:Measure)

zeroMeasure(m:Measure):Result ==
a := coerce(0$F$)\texttt{AnyFunctions1(F)}
text := coerc("No available routine appears appropriate")\texttt{AnyFunctions1(ST)}
r := construct([[result\texttt{Symbol},a],[method\texttt{Symbol},text]])\texttt{Result}
concat(measure2Result m,r)$ExpertSystemToolsPackage

measureSpecific(name:ST,R:RT,p:PDEB):Record(measure:F,explanations:ST) ==
name = "d03eefAnnaType" => measure(R,p)$d03eefAnnaType
--name = "d03fafAnnaType" => measure(R,p)$d03fafAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions

measure(P: NumericalPDEProblem,R:RT): Measure ==
p:PDEB := retract(P)$NumericalPDEProblem
sofar := 0$F
best := "none" :: ST
routs := copy R
routs := selectPDERoutines(routs)$RT
empty?(routs)$RT =>
  error("measure", "no routines found")$ErrorFunctions
rout := inspect(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
meth := empty()$LST
for i in 1..# routs repeat
  rout := extract!(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
n := e.domainName
if e.defaultMin > sofar then
  m := measureSpecific(n,R,p)
  if m.measure > sofar then
    sofar := m.measure
    best := n
    str:LST := [string(rout.key)$Symbol "measure: "
    outputMeasure(m.measure)$ExpertSystemToolsPackage " - "
    m.explanations]
  else
    str := [string(rout.key)$Symbol " is no better than other routines"]
meth := append(meth,str)$LST
[sofar,best,meth]

measure(P: NumericalPDEProblem): Measure == measure(P,routines()$RT)

solveSpecific(p:PDEB,n:ST): Result ==
n = "d03eefAnnaType" => PDESolve(p)$d03eefAnnaType
--n = "d03fafAnnaType" => PDESolve(p)$d03fafAnnaType
error("solveSpecific","invalid type name: " n)$ErrorFunctions

changeName(ans:Result,name:ST): Result ==
sy:Symbol := coerce(name "Answer")$Symbol
anyAns:Any := coerce(ans)$AnyFunctions1(Result)
construct([[sy,anyAns]])$Result

recoverAfterFail(p:PDEB,routs:RT,m:Measure,iint:Integer,r:Result):
  Record(a:Result,b:Measure) ==
while positive?(iint) repeat
PACKAGE PDEPACK ANNAPARTIALDIFFERENTIALEQUATIONPACKAGE

routineName := m.name
s := recoverAfterFail(routs,routineName(1..6),iint)$RT
s case "failed" => iint := 0
(a = "no action")@Boolean => iint := 0
fl := coerce(s)$AnyFunctions1(ST)
flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
m2 := measure(p::NumericalPDEProblem,routs)
zero?(m2.measure) => iint := 0
r2:Result := solveSpecific(p,m2.name)
m := m2
insert!(flrec,r2)$Result
r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
iany := search(ifail@Symbol,r2)$Result
iany case "failed" => iint := 0
iint := retract(iany)$AnyFunctions1(Integer)
[r,m]

solve(P:NumericalPDEProblem,t:RT):Result ==
routs := copy(t)$RT
m := measure(P,routs)
p:PDEB := retract(P)$NumericalPDEProblem
zero?(m.measure) => zeroMeasure m
r := solveSpecific(p,n := m.name)
iany := search(ifail@Symbol,r)$Result
iint := 0$Integer
if (iany case Any) then
  iint := retract(iany)$AnyFunctions1(Integer)
if positive?(iint) then
  tu:Record(a:Result,b:Measure) := recoverAfterFail(p,routs,m,iint,r)
r := tu.a
m := tu.b
expl := getExplanations(routs,n(1..6))$RoutinesTable
expla := coerce(expl)$AnyFunctions1(LST)
explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
r := concat(construct([explaa]),r)
concat(measure2Result m,r)$ExpertSystemToolsPackage

solve(P:NumericalPDEProblem):Result == solve(P,routines()$RT)
  LEF,s:ST,to:DF):Result ==
  cx:PDEC := [f2df xmi, f2df xma, nx, 1, empty()$MDF, empty()$MDF]
cy:PDEC := [f2df ymi, f2df yma, ny, 1, empty()$MDF, empty()$MDF]
p:PDEB := [[ef2edf e for e in pe],[cx,cy],
  [ef2edf u for u in w] for w in bo],s,to)
solve(p::NumericalPDEProblem,routines()$RT)

  LEF,s:ST):Result ==
solve(xmi,xma,ymi,yma,nx,ny,pe,bo,s,0.0001::DF)
package ANY1 AnyFunctions1

AnyFunctions1 implements several utility functions for working with Any. These functions are used to go back and forth between objects of Any and objects of other types.

See Also:
  o )show AnyFunctions1
AnyFunctions1 (ANY1)

Exports:
  coerce retract retractable? retractIfCan

— package ANY1 AnyFunctions1 —

)abbrev package ANY1 AnyFunctions1
++ Description:
++ \spadtype{AnyFunctions1} implements several utility functions for
++ working with \spadtype{Any}. These functions are used to go back
++ and forth between objects of \spadtype{Any} and objects of other
++ types.

AnyFunctions1(S:Type): with
  coerce : S -> Any
    ++ coerc(s) creates an object of \spadtype{Any} from the
    ++ object \spad{s} of type \spad{S}.
  retractIfCan: Any -> Union(S, "failed")
    ++ retract(a) tries change \spad{a} into an object
    ++ of type \spad{S}. If it can, then such an object is
    ++ returned. Otherwise, "failed" is returned.
  retractable?: Any -> Boolean
    ++ retractable?(a) tests if \spad{a} can be converted
    ++ into an object of type \spad{S}.
  retract : Any -> S
    ++ retract(a) tries to convert \spad{a} into an object of
    ++ type \spad{S}. If possible, it returns the object.
    ++ Error: if no such retraction is possible.

== add
  import NoneFunctions1(S)
Sexpr:SExpression := devaluate(S)$Lisp

retractable? a == dom(a) = Sexpr
coerce(s:S):Any == any(Sexpr, s::None)

retractIfCan a ==
  retractable? a => obj(a) pretend S
  "failed"

retract a ==
  retractable? a => obj(a) pretend S
  error "Cannot retract value."

package API ApplicationProgramInterface

getDomains 'Collection
--R
--R (1)
--R {AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList,
--R DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet,
--R GeneralSparseTable, GeneralTriangularSet, HashTable, IndexedBits,
--R IndexedFlexibleArray, IndexedList, IndexedOneDimensionalArray,
--R IndexedString, IndexedVector, InnerTable, KeyedAccessFile, Library, List,
The image contains a page from a document discussing various packages in a programming language, specifically Axiom. The text includes a list of types such as `ListMultiDictionary`, `Multiset`, `NeitherSparseOrDensePowerSeries`, `OneDimensionalArray`, `Point`, `PrimitiveArray`, `RegularChain`, `RegularTriangularSet`, `Result`, `RoutinesTable`, `Set`, `SparseTable`, `SquareFreeRegularTriangularSet`, `Stream`, `String`, `StringTable`, `Table`, `U16Vector`, `U32Vector`, `U8Vector`, `Vector`, `WuWenTsunTriangularSet`. It also mentions operations like `difference`, `credits()`, and `Type: Set(Symbol)`. The document also includes a list of contributors to Axiom.
The summary command simply types out the contents of the file $AXIOM/lib/summary so no output is captured in the spool file.

--- ApplicationProgramInterface.input ---

...)show API
---R ApplicationProgramInterface is a package constructor
---R Abbreviation for ApplicationProgramInterface is API
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for API
---R
---R------------------------------ Operations ------------------------------
---R credits : () -> Void
---R getDomains : Symbol -> Set(Symbol)
---R summary : () -> Void
---R getAncestors : Symbol -> Set(Symbol)
---E 5

...)show API
---R ApplicationProgramInterface is a package constructor
---R Abbreviation for ApplicationProgramInterface is API
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for API
---R
---R------------------------------ Operations ------------------------------
---R credits : () -> Void
---R getDomains : Symbol -> Set(Symbol)
---R summary : () -> Void
---R getAncestors : Symbol -> Set(Symbol)
---R
---E 5

...)show API
---R ApplicationProgramInterface is a package constructor
---R Abbreviation for ApplicationProgramInterface is API
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for API
---R
---R------------------------------ Operations ------------------------------
---R credits : () -> Void
---R getDomains : Symbol -> Set(Symbol)
---R summary : () -> Void
---R getAncestors : Symbol -> Set(Symbol)
---R
---E 5
reportInstantiations(true)
--R
--R Type: Void
--R# instantiated/# dropped/domain name
--R------------------------------------
--R Totals: 0 instantiated
--R 0 inside coerceInteractive
--R 0 inside canCoerceFrom
--R 0 inside evalMmCond
--R 0 reinstantiated
--R 0 dropped
--R 0 distinct domains instantiated/dropped
--E 7

--S 8 of 9
1
--R
--R
--R (7) 1
--R Type: PositiveInteger
--R# instantiated/# dropped/domain name
--R------------------------------------
--R1 0 PositiveInteger
--R1 0 NonNegativeInteger
--R Totals: 2 instantiated
--R 0 inside coerceInteractive
--R 0 inside canCoerceFrom
--R 0 inside evalMmCond
--R 0 reinstantiated
--R 0 dropped
--R 2 distinct domains instantiated/dropped
--E 8

--S 9 of 9
reportInstantiations(true)
--R
--R Type: Void
--E 9

)spool
)lisp (bye)

— ApplicationProgramInterface.help —

====================================================================
ApplicationProgramInterface examples
====================================================================
This package contains useful functions that expose Axiom system internals.

The ApplicationProgramInterface exposes Axiom internal functions which might be useful for understanding, debugging, or creating tools.

The getDomains function takes the name of a category and returns a set of domains which inherit from that category:

\[
\text{getDomains 'Collection}
\]

\{AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList, DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet, GeneralSparseTable, GeneralTriangularSet, HashTable, IndexedBits, IndexedFlexibleArray, IndexedList, IndexedOneDimensionalArray, IndexedString, IndexedVector, InnerTable, KeyedAccessFile, Library, List, ListMultiDictionary, Multiset, NeitherSparseOrDensePowerSeries, OneDimensionalArray, Point, PrimitiveArray, RegularChain, RegularTriangularSet, Result, RoutinesTable, Set, SparseTable, SquareFreeRegularTriangularSet, Stream, String, StringTable, Table, U32Vector, Vector, WuWenTsunTriangularSet\}

Type: Set(Symbol)

This can be used to form the set-difference of two categories:

\[
\text{difference(getDomains 'IndexedAggregate, getDomains 'Collection)}
\]

\{DirectProduct, DirectProductMatrixModule, DirectProductModule, HomogeneousDirectProduct, OrderedDirectProduct, SplitHomogeneousDirectProduct\}

Type: Set(Symbol)

The credits function prints a list of the people who have contributed to the development of Axiom. This is equivalent to the )credits command.

The summary function prints a short list of useful console commands.

The getAncestors function takes the name of a domain or category and returns a list of categories from which it inherits.

\[
\text{getAncestors 'IndexedAggregate}
\]

\{Aggregate, BasicType, CoercibleTo, Eltable, EltableAggregate, Evalable, HomogeneousAggregate, InnerEvalable, SetCategory, Type\}

Type: Set(Symbol)

The reportInstantiations function information about what domains are instantiated by an expression. In a clean Axiom you'll see
(1) -> reportInstantiations(true)

# instantiated/# dropped/domain name
------------------------------------
1 0 Void
1 0 String
1 0 SingleInteger
1 0 PrimitiveArray(OutputForm)
1 0 OutputForm
1 0 List(OutputForm)
1 0 Integer
1 0 Character
Totals: 8 instantiated
   0 inside coerceInteractive
   0 inside canCoerceFrom
   0 inside evalMmCond
   0 reinstantiated
   0 dropped
   8 distinct domains instantiated/dropped

Which shows that 8 domains were instantiated.
If a new domain is requested, more will be instantiated.

(2) -> 1

(2) 1

# instantiated/# dropped/domain name
------------------------------------
1 0 PositiveInteger
1 0 NonNegativeInteger
Totals: 2 instantiated
   0 inside coerceInteractive
   0 inside canCoerceFrom
   0 inside evalMmCond
   0 reinstantiated
   0 dropped
   2 distinct domains instantiated/dropped

However, this happens only once. Invoking already instantiated
domains does nothing since they already exist.

(3) -> 1

(3) 1

# instantiated/# dropped/domain name
------------------------------------
Totals: 0 instantiated
   0 inside coerceInteractive
0 inside canCoerceFrom
0 inside evalMmCond
0 reinstantiated
0 dropped
0 distinct domains instantiated/dropped

We call the function with false to turn off this information.

(4) -> reportInstantiations(false)

Type: Void

ApplicationProgramInterface (API)

Exports:

— package API ApplicationProgramInterface —

)abbrev package API ApplicationProgramInterface
++ Author: Timothy Daly, Martin Rubey
++ Date Created: 3 March 2009
++ Date Last Updated: 24 March 2012
++ Description:
++ This package contains useful functions that expose Axiom system internals

ApplicationProgramInterface(): Exports == Implementation where
Exports ==> with
  getDomains : Symbol -> Set Symbol
  ++ The getDomains(s) takes a category and returns the list of domains
  ++ that have that category
  ++
++X getDomains 'IndexedAggregate
getAncestors : Symbol -> Set Symbol
  ++ The getAncestor(s) takes a category and returns the list of domains
  ++ that have that category as ancestors
  ++
++X getAncestors 'IndexedAggregate

credits : () -> Void
  ++ credits() prints a list of people who contributed to Axiom
  ++
++X credits()

summary : () -> Void
  ++ summary() prints a short list of useful console commands
  ++
++X summary()

reportInstantiations: Boolean -> Void
  ++ A debugging tool to show instantiation information
  ++
++X reportInstantiations(true)
++X 1
++X reportInstantiations(false)

Implementation ==> add
getDomains(cat:Symbol):Set(Symbol) ==
  set [symbol car first destruct a _
    for a in (destruct domainsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

getAncestors(cat:Symbol):Set(Symbol) ==
  set [symbol car first destruct a _
    for a in (destruct ancestorsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

credits() == ( credits()$Lisp ; void() )

summary() == ( summary()$Lisp ; void() )

reportInstantiations(b:Boolean): Void ==
REPORTINSTANTIATIONS(b)$Lisp
void

— API.dotabb —

"API" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPRULE"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"API" -> "ORDSET"
package APPRULE ApplyRules

--- ApplyRules.input ---

)set break resume
)sys rm -f ApplyRules.output
)spool ApplyRules.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ApplyRules
--E 1

)spool
)lisp (bye)

---

--- ApplyRules.help ---

====================================================================
ApplyRules examples
====================================================================

This package apply rewrite rules to expressions, calling the pattern matcher.

See Also:
o )show ApplyRules

---
ApplyRules (APPRULE)

Exports:

applyRules  localUnquote

— package APPRULE ApplyRules —

)abbrev package APPRULE ApplyRules
++ Applications of rules to expressions
++ Author: Manuel Bronstein
++ Date Created: 20 Mar 1990
++ Date Last Updated: 5 Jul 1990
++ Description:
++ This package apply rewrite rules to expressions, calling
++ the pattern matcher.

ApplyRules(Base, R, F): Exports == Implementation where
Base : SetCategory
R : Join(Ring, PatternMatchable Base, OrderedSet,
    ConvertibleTo Pattern Base)
F : Join(FunctionSpace R, PatternMatchable Base,
    ConvertibleTo Pattern Base)

P ==> Pattern Base
PR ==> PatternMatchResult(Base, F)
RR ==> RewriteRule(Base, R, F)
K ==> Kernel F

Exports ==> with
    applyRules : (List RR, F) -> F
    ++ applyRules([r1,...,rn], expr) applies the rules
    ++ r1,...,rn to f an unlimited number of times, i.e. until
    ++ none of r1,...,rn is applicable to the expression.
    applyRules : (List RR, F, PositiveInteger) -> F
    ++ applyRules([r1,...,rn], expr, n) applies the rules
    ++ r1,...,rn to f a most n times.
    localUnquote: (F, List Symbol) -> F
++ localUnquote(f,ls) is a local function.

Implementation ==> add
import PatternFunctions1(Base, F)

splitRules : List RR -> Record(lker: List K,lval: List F,rl: List RR)
localApply : (List K, List F, List RR, F, PositiveInteger) -> F
rewrite : (F, PR, List Symbol) -> F
app : (List RR, F) -> F
applist : (List RR, List F) -> List F
isit : (F, P) -> PR
isitwithpred: (F, P, List P, List PR) -> PR

applist(lrule, arglist) == [app(lrule, arg) for arg in arglist]

splitRules l ==
ncr := empty()$List(RR)
lk := empty()$List(K)
lv := empty()$List(F)
for r in l repeat
   if (u := retractIfCan(r)@Union(Equation F, "failed"))
      case "failed" then ncr := concat(r, ncr)
      else
         lk := concat(retract(lhs(u::Equation F))@K, lk)
         lv := concat(rhs(u::Equation F), lv)
   [lk, lv, ncr]

applyRules(l, s) ==
rec := splitRules l
repeat
   (new:= localApply(rec.lker,rec.lval,rec.rl,s,1)) = s => return s
   s := new
applyRules(l, s, n) ==
rec := splitRules l
localApply(rec.lker, rec.lval, rec.rl, s, n)

localApply(lk, lv, lrule, subject, n) ==
for i in 1..n repeat
   for k in lk for v in lv repeat
      subject := eval(subject, k, v)
      subject := app(lrule, subject)
subject

rewrite(f, res, l) ==
lk := empty()$List(K)
lv := empty()$List(F)
for rec in destruct res repeat
   lk := concat(kernel(rec.key), lk)
   lv := concat(rec.entry, lv)
localUnquote(eval(f, lk, lv), l)

if R hasConvertibleTo InputForm then
    localUnquote(f, l) ==
    empty? l => f
    eval(f, l)
else
    localUnquote(f, l) == f

isitwithpred(subject, pat, vars, bad) ==
    failed?(u := patternMatch(subject, pat, new()$PR)) => u
    satisfy?(u, pat)::Boolean => u
    member?(u, bad) => failed()
    for v in vars repeat addBadValue(v, getMatch(v, u)::F)
    isitwithpred(subject, pat, vars, concat(u, bad))

isit(subject, pat) ==
    hasTopPredicate? pat =>
    for v in (l := variables pat) repeat resetBadValues v
    isitwithpred(subject, pat, l, empty())
    patternMatch(subject, pat, new()$PR)

app(lrule, subject) ==
    for r in lrule repeat
        not failed?(u := isit(subject, pattern r)) =>
        return rewrite(rhs r, u, quotedOperators r)
    (k := retractIfCan(subject)@Union(K, "failed")) case K =>
        operator(k::K) applist(lrule, argument(k::K))
    (l := isPlus subject) case List(F) => +/applist(lrule,l::List(F))
    (l := isTimes subject) case List(F) => */applist(lrule,l::List(F))
    (e := isPower subject) case Record(val:F, exponent:Integer) =>
        ee := e::Record(val:F, exponent:Integer)
        f := app(lrule, ee.val)
        positive?(ee.exponent) => f ** (ee.exponent)::NonNegativeInteger
        recip(f)::F ** (- ee.exponent)::NonNegativeInteger
        subject

—— APPRULE.dotabb ——

"APPRULE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPRULE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"APPRULE" -> "FS"
package APPLYORE ApplyUnivariateSkewPolynomial

--- ApplyUnivariateSkewPolynomial.input ---

)set break resume
)sys rm -f ApplyUnivariateSkewPolynomial.output
)spool ApplyUnivariateSkewPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ApplyUnivariateSkewPolynomial
--E 1

)spool
)lisp (bye)

---

--- ApplyUnivariateSkewPolynomial.help ---

====================================================================
ApplyUnivariateSkewPolynomial examples
====================================================================

ApplyUnivariateSkewPolynomial (internal) allows univariate skew polynomials to be applied to appropriate modules.

See Also:
o )show ApplyUnivariateSkewPolynomial

---
ApplyUnivariateSkewPolynomial (APPLYORE)

Exports:
apply

— package APPLYORE ApplyUnivariateSkewPolynomial —

)abbrev package APPLYORE ApplyUnivariateSkewPolynomial
++ Author: Manuel Bronstein
++ Date Created: 7 December 1993
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{ApplyUnivariateSkewPolynomial} (internal) allows univariate
++ skew polynomials to be applied to appropriate modules.

ApplyUnivariateSkewPolynomial(R:Ring, M: LeftModule R,
P: UnivariateSkewPolynomialCategory R): with
apply: (P, M -> M, M) -> M
  ++ apply(p, f, m) returns \spad{p(m)} where the action is given
  ++ by \spad{x m = f(m)}.
  ++ \spad{f} must be an R-pseudo linear map on M.
== add
apply(p, f, m) ==
w:M := 0
mn:M := m
for i in 0..degree p repeat
  w := w + coefficient(p, i) * mn
  mn := f mn
w

— APPLYORE.dotabb —

"APPLYORE" [color=#FF4488,href="bookvol10.4.pdf#nameddest=APPLYORE"]
package ASSOEQ AssociatedEquations

--- AssociatedEquations.input ---

)set break resume
)sys rm -f AssociatedEquations.output
)spool AssociatedEquations.output
)set message test on
)set message auto off
)clear all

--- 1 of 1
)show AssociatedEquations
--- E 1

)spool
)lisp (bye)

---

--- AssociatedEquations.help ---

====================================================================
AssociatedEquations examples
====================================================================

AssociatedEquations provides functions to compute the associated
equations needed for factoring operators

See Also:
o )show AssociatedEquations

---
AssociatedEquations (ASSOCEQ)

Exports:

- associatedEquations
- associatedSystem
- uncouplingMatrices

--- package ASSOCEQ AssociatedEquations ---

)abbrev package ASSOCEQ AssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 10 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{AssociatedEquations} provides functions to compute the
++ associated equations needed for factoring operators

AssociatedEquations(R, L):Exports == Implementation where
R: IntegralDomain
L: LinearOrdinaryDifferentialOperatorCategory R

PI ==> PositiveInteger
N ==> NonNegativeInteger
MAT ==> Matrix R
REC ==> Record(minor: List PI, eq: L, minors: List List PI, ops: List L)

Exports ==

- associatedSystem: (L, PI) -> Record(mat: MAT, vec:Vector List PI)
  ++ associatedSystem(op, m) returns \spad{[M, w]} such that the
  ++ m-th associated equation system to L is \spad{w' = M w}.
- uncouplingMatrices: MAT -> Vector MAT
  ++ uncouplingMatrices(M) returns \spad{[A_1, \ldots, A_n]} such that if
  ++ \spad{[y = [y_1, \ldots, y_n]} is a solution of \spad{y' = M y}, then
  ++ \spad{[y_j', y_j'', \ldots, y_j^{(n)}]} = \spad{A_j y} for all j's.
  if R has Field then
  ++ associatedEquations: (L, PI) -> REC
  ++ associatedEquations(op, m) returns \spad{[w, eq, lw, lop]}
  ++ such that \spad{eq(w) = 0} where w is the given minor, and
  ++ \spad{lw_i = lop_i(w)} for all the other minors.
Implementation ==> add

`makeMatrix: (Vector MAT, N) -> MAT`

diff:L := D()

`makeMatrix(v, n) == matrix [parts row(v.i, n) for i in 1..#v]`

`associatedSystem(op, m) ==
  eq: Vector R
  S := SetOfMIntegersInOneToN(m, n := degree(op)::PI)
  w := enumerate()$S
  s := size()$S
  ww:Vector List PI := new(s, empty())
  M:MAT := new(s, s, 0)
  m1 := (m::Integer - 1)::PI
  an := leadingCoefficient op
  a:Vector(R) := [- (coefficient(op, j) exquo an)::R for j in 0..n - 1]
  for i in 1..s repeat
    eq := new(s, 0)
    wi := w.i
    ww.i := elements wi
    for k in 1..m1 repeat
      u := incrementKthElement(wi, k::PI)$S
      if u case S then eq(lookup(u::S)) := 1
      if member?(n, wi) then
        for j in 1..n | a.j ^= 0 repeat
          u := replaceKthElement(wi, m, j::PI)
          if u case S then
            eq(lookup(u::S)) := (odd? delta(wi, m, j::PI) => -a.j; a.j)
      else
        u := incrementKthElement(wi, m)$S
        if u case S then eq(lookup(u::S)) := 1
      setRow_!(M, i, eq)
  [M, ww]

uncouplingMatrices m ==
  n := nrows m
  v:Vector MAT := new(n, zero(1, 0)$MAT)
  v.1 := mi := m
  for i in 2..n repeat v.i := mi := map((z1:R):R +-> diff z1, mi) + mi * m
  [makeMatrix(v, i) for i in 1..n]

if R has Field then
  import PrecomputedAssociatedEquations(R, L)

`makeop: Vector R -> L`

`makeeq: (Vector List PI, MAT, N, N) -> REC`

`computeIt: (L, PI, N) -> REC`
makeeq(v, m, i, n) ==
[v.i, makeop row(m, i) - 1, [v.j for j in 1..n | j ^= i],
 [makeop row(m, j) for j in 1..n | j ^= i]]

associatedEquations(op, m) ==
(u := firstUncouplingMatrix(op, m)) case "failed" => computeIt(op,m,1)
(v := inverse(u::MAT)) case "failed" => computeIt(op, m, 2)
S := SetOfMIntegersInOneToN(m, degree(op)::PI)
w := enumerate()$

s := size()$
ww:Vector List PI := new(s, empty())
for i in 1..s repeat ww.i := elements(w.i)
makeeq(ww, v::MAT, 1, s)

computeIt(op, m, k) ==
rec := associatedSystem(op, m)
a := uncouplingMatrices(rec.mat)
n := #a
for i in k..n repeat
  (u := inverse(a.i)) case MAT => return makeeq(rec.vec,u::MAT,i,n)
error "associatedEquations: full degenerate case"

makeop v ==
op:L := 0
for i in 1..#v repeat op := op + monomial(v i, i)
op


— ASSOCEQ.dotabb —

"ASSOCEQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ASSOCEQ"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ASSOCEQ" -> "IVECTOR"

— AttachPredicates.input —

)set break resume
)sys rm -f AttachPredicates.output
)spool AttachPredicates.output
---
— AttachPredicates.help —

====================================================================
AttachPredicates examples
====================================================================

Attaching predicates to symbols for pattern matching.

See Also:
  o )show AttachPredicates

---

AttachPredicates (PMPRED)

Exports:
  suchThat

    — package PMPRED AttachPredicates —
abbrev package PMPRED AttachPredicates
++ Predicates for pattern-matching
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching predicates to symbols for pattern matching.

AttachPredicates(D:Type): Exports == Implementation where
FE ==> Expression Integer

Exports ==> with
   suchThat: (Symbol, D -> Boolean) -> FE
      ++ suchThat(x, foo) attaches the predicate foo to x.
suchThat: (Symbol, List(D -> Boolean)) -> FE
      ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
         ++ f1 and f2 and ... and fn to x.

Implementation ==> add
   import FunctionSpaceAttachPredicates(Integer, FE, D)

   suchThat(p:Symbol, f:D -> Boolean)  == suchThat(p::FE, f)
   suchThat(p:Symbol, l:List(D -> Boolean)) == suchThat(p::FE, l)

— PMPRED.dotabb —
"PMPRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPRED"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PMPRED" -> "PID"
"PMPRED" -> "OAGROUP"

package AXSERV AxiomServer

— AxiomServer.input —

)set break resume
)sys rm -f AxiomServer.output
)spool AxiomServer.output
)set message test on
This package provides functions to support a web server for the new Axiom Browser functions.

See Also:
- )show AxiomServer

Exports:
- multiServ
- axServer
- getDatabase
The axServer function handles the socket connection on the given port. When it gets a input on the socket it calls the server function on the socket input.

— package AXSERV AxiomServer —

axServer(port:Integer,serverfunc:SExpression->Void):Void ==
  WriteLine$Lisp "$listening on port 8085"
  s := SiSock(port,serverfunc)$ Lisp
  -- To listen for just one connection and then close the socket
  -- uncomment i := 0.
The multiServ function parses the socket input. It expects either a GET or POST request. A GET request fetches a new page, calling “getFile”. A POST request starts with

- “command=” which expects axiom interpreter commands. When this is recognized we call the “getCommand” function.
- “lispcall=” which expects lisp interpreter input. When this is recognized we call the “getLisp” function.

---

--- package AXSERV AxiomServer ---

multiServ(s:SExpression):Void ==
-- WriteLine("multiServ begin")$Lisp
headers:String := ""
char:String
-- read in the http headers
while (char := STRING(READ_-CHAR_-NO_-HANG(s,NIL$Lisp,'EOF)$Lisp)$Lisp) ^= "EOF"_ repeat
  headers := concat [headers,char]
-- sayTeX$Lisp headers
StringMatch("(\[^ \]*)", headers)$Lisp
u:UniversalSegment(Integer)
u := segment(MatchBeginning(1)$Lisp+1, MatchEnd(1)$Lisp)$UniversalSegment(Integer)
reqtype:String := headers.u
-- sayTeX$Lisp concat ['request type: ",reqtype]
if reqtype = "GET" then
  StringMatch("GET ([^ ])\", headers)$Lisp
  u:UniversalSegment(Integer)
u := segment(MatchBeginning(1)$Lisp+1, MatchEnd(1)$Lisp)$UniversalSegment(Integer)
  getFile(s,headers,u)
if reqtype = "POST" and StringMatch("command=.*,\", headers)$Lisp > 0 then
  u:UniversalSegment(Integer)
u := segment(MatchBeginning(1)$Lisp+1,
MatchEnd(1)$Lisp)$UniversalSegment(Integer)
getCommand(s,headers.u)
if reqtype = "POST" and StringMatch("interpcall=(.*)",headers)$Lisp > 0 then
  u:UniversalSegment(Integer)
  u := segment(MatchBeginning(1)$Lisp+1,_,
    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
getInterp(s,headers.u)
if reqtype = "POST" and StringMatch("lispcall=(.*)",headers)$Lisp > 0 then
  u:UniversalSegment(Integer)
  u := segment(MatchBeginning(1)$Lisp+1,_,
    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
getLisp(s,headers.u)
if reqtype = "POST" and StringMatch("showcall=(.*)",headers)$Lisp > 0 then
  u:UniversalSegment(Integer)
  u := segment(MatchBeginning(1)$Lisp+1,_,
    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
getShow(s,headers.u)

-- WriteLine("multiServ end")$Lisp
-- WriteLine(""")$Lisp

getFile

Given a socket and the URL of the file we create an input stream that contains the file. If the
filename contains a question mark then we need to parse the parameters and dynamically
construct the file contents.

— package AXSERV AxiomServer —

g getFile(s:SExpression,pathvar:String):Void ==
  WriteLine(""")$Lisp
  WriteLine$Lisp concat ["getFile: ",pathvar]
  params:=split(pathvar,char ":")
  if #params = 1
    then if not null? PATHNAME_NAME(PATHNAME(pathvar)$Lisp)$Lisp
      then
        contentType:String := getContentType(pathvar)
        q:=Open(pathvar)$Lisp
        if null? q
          then
            q := MAKE_STRING_INPUT_STREAM(_
              makeErrorPage("File doesn't exist"))$Lisp
          else
            q:=MAKE_STRING_INPUT_STREAM(_
              makeErrorPage("Problem with file path"))$Lisp


else
   q:=MAKE_-STRING_-INPUT_-STREAM(makeDBPage(pathvar))$Lisp
outputToSocket(s,readTheFile(q),contentType)

---

makeErrorPage

    — package AXSERV AxiomServer —

makeErrorPage(msg:String):String ==
   page:String:="<!DOCTYPE html PUBLIC "
   page:=page "-//W3C//DTD XHTML 1.0 Strict//EN" "
   page:=page "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
   page:=page "<html xmlns="http://www.w3.org/1999/xhtml">
   page:=page "<head><title>Error</title></head><body>
   page:=page "<p>" msg "</p></body></html>
   -- WriteLine(page)$Lisp
   page

---

getDescription

We need to fish around in the data structure to return the piece of documentation for the domain. We have to call the lisp version of GETDATABASE because the version above returns a string object. The string object is missing quotes and cannot be properly read. So we need to get the lisp object and work with it in native form first.
The doc string also contains spad markup which we need to replace with html.

    — package AXSERV AxiomServer —

getDescription(dom:String):String ==
   d:=CADR(CADAR(GETDATABASE(INTERN(dom)$Lisp,'DOCUMENTATION)$Lisp)$Lisp)$Lisp
   string d

---

ggetSourceFile

During build we construct a hash table that takes the chunk name as the key and returns the filename. We reconstruct the chunk name here and do a lookup for the source file.

    — package AXSERV AxiomServer —

getSourceFile(constructorkind:String,_
    abbreviation:String,_

dom: String) : String ==
  sourcekey: = "@<<" constructorkind " " abbreviation " " dom ">>"
  -- WriteLine(sourcekey)$Lisp
  sourcefile:=lowerCase last split(getDatabase(dom, "SOURCEFILE"), char "/")
  sourcefile:=sourcefile ".pamphlet"

makeDBPage

— package AXSERV AxiomServer —

makeDBPage(pathvar: String) : String ==
  params: List(String) := split(pathvar, char "?")
  for i in 1..#params repeat WriteLine$Lisp concat ["params: ", params.i]
  pathparts: List(String) := split(params.1, char "/")
  for i in 1..#pathparts repeat
    WriteLine$Lisp concat ["pathparts: ", pathparts.i]
  pagename:=last pathparts
  WriteLine$Lisp concat ["pagename: ", pagename]
  cmd:=first split(pagename, char ".")
  WriteLine$Lisp concat ["cmd: ", cmd]
  args: List(String) := split(params.2, char "&")
  for i in 1..#args repeat WriteLine$Lisp concat ["args: ", args.i]
  page:="<!DOCTYPE html PUBLIC "
  page:=page "-//W3C//DTD XHTML 1.0 Strict//EN_ "
  page:=page "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd_">"
  page:=page "<html xmlns=_"http://www.w3.org/1999/xhtml_">"
  page:=page "<head>
  page:=page "<meta http-equiv=_"Content-Type_" content=_"text/html_"
  page:=page " charset=_us-ascii_"/>
  page:=page "<title>_ cmd _ args.1 _</title"></head>
  page:=page "<style> html { background-color: #ECEA81; } </style>"
  page:=page "<body>
  cmd = "db" =>
    dom:=args.1
    domi:=INTERN(dom)$Lisp
    -- category, domain, or package?
    constructorkind:=getDatabase(dom, "CONSTRUCTORKIND")
    abbreviation:=getDatabase(dom, "ABBREVIATION")
    sourcefile:=getDatabase(dom, "SOURCEFILE")
    constructorkind.1:=upperCase constructorkind.1
    description:=getDescription(dom)
  page:=page "<div align=_"center_">"
  page:=page "<img align=_"middle_" src=_"doctitle.png_"/></div><hr/>
  page:=page "<div align=_"center_">" constructorkind _ dom _</div><hr/>
  page:=page "<table>"
package axserv axiomserver

<TR><TD valign="top">Description: </TD></TR>
<TR><TD>Abbreviation: </TD><TD>abbreviation "</TD></TR>
<TR><TD>Source File: </TD><TD>sourcefile "</TD></TR>
</table>

<a href="?dom=Ancestors">Ancestors</a>

<a href="?dom=Dependents">Dependents</a>

<a href="?dom=Exports">Exports</a>

<a href="?dom=Parents">Parents</a>

<a href="?dom=Users">Users</a>

<a href="?dom=Attributes">Attributes</a>

<a href="?dom=Examples">Examples</a>

<a href="?dom=Operations">Operations</a>

<a href="?dom=SearchPath">Search Path</a>

<a href="?dom=Uses">Uses</a>

cmd = "op" =>

dom:=INTERN(dom)$Lisp
-- category, domain, or package?

constructorkind:=getDatabase(dom,"CONSTRUCTORKIND")
abbreviation:=getDatabase(dom,"ABBREVIATION")
sourcefile:=getDatabase(dom,"SOURCEFILE")
constructorkind.1:=toUpperCase constructorkind.1
description:=getDescription(dom)
CHAPTER 2. CHAPTER A

```html
page = "<div align="center">"
page = "<img align="middle" src="doctitle.png"></div>"<hr/>
page = "<div align="center">" constructorkind " " dom "</div><hr/>
page = "<table>
page = "<tr><td valign="top">Description: </td><td>" description "</td></tr>
page = "<tr><td>Abbreviation: </td><td>" abbreviation "</td></tr>
page = "<tr><td>Source File: </td><td>" sourcefile "</td></tr>
page = "</table>"
page = "<table>
page = "<tr>
page = "<td>
page = "<a href="?" dom "&lookup=Ancestors">Ancestors</a>"
page = "</td>
page = "<td>
page = "<a href="?" dom "&lookup=Dependents">Dependents</a>"
page = "</td>
page = "<td>
page = "<a href="?" dom "&lookup=Exports">Exports</a>"
page = "</td>
page = "<td>
page = "<a href="?" dom "&lookup=Parents">Parents</a>"
page = "</td>
page = "<td>
page = "<a href="?" dom "&lookup=Users">Users</a>"
page = "</td>
page = "</tr>
page = "</table>
```

```
```
readTheFile

We have q which is a stream which contains the file. We read the file into a string-stream to get it all into one string. We return the string.

    — package AXSERV AxiomServer —

    readTheFile(q:SExpression):String ==
      r := MAKE_-STRING_-OUTPUT_-STREAM()
      SiCopyStream(q,r)
      filestream:String := GET_-OUTPUT_-STREAM_-STRING(r)
      CLOSE(r)
      CLOSE(q)
      filestream

outputToSocket

We have “s” which is the socket, “filestream” which is the text of the file to output, and “contentType” which is the HTML Content-Type. We construct the HTML header information according to the standard and prepend it to the file. The resulting string is output to the socket.

    — package AXSERV AxiomServer —

    outputToSocket(s:SExpression,filestream:String,contentType:String):Void ==
      filelength:String := string(#filestream)
      file:String := ""
      nl:STRING(NewLine$Lisp)
      file := concat ["Content-Length: ",filelength,nl,nl,file]
      file := concat ["Connection: close",nl,file]
      file := concat ["Content-Type: ",contentType,nl,file]
      file := concat ["HTTP/1.1 200 OK",nl,file]
      file := concat [file,filestream]
      f:=MAKE_-STRING_-INPUT_-STREAM(file)$Lisp
      SiCopyStream(f,s)$Lisp
      CLOSE(f)$Lisp
      CLOSE(s)$Lisp
getCommand

The getCommand function is invoked when the HTTP request is a POST and contains the string "command". Essentially the game here is to rebind the various output streams used by Axiom so we can capture the normal output. This function returns a set of HTML 5 div blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

```lisp
package AXSERV AxiomServer

getCommand(s:SExpression,command:String):Void ==
  WriteLine$Lisp concat ["getCommand: ",command]
  SETQ(tmpmathml$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
  SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
  SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
  SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
  SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
  SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
  ans := string parseAndEvalToStringEqNum$Lisp command
  SETQ(resultmathml$Lisp, GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)$Lisp
  SETQ(resultalgebra$Lisp, GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
  SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
  SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
  CLOSE(tmpmathml$Lisp)$Lisp
  CLOSE(tmpalgebra$Lisp)$Lisp
  -- Since strings returned from axiom are going to be displayed in html I
  -- should really check for the characters &,<,> and replace them with
  -- &amp;,&lt;,&gt;.
  -- At present I only check for ampersands in formatMessages.
  mathml:String := string(resultmathml$Lisp)
  algebra:String := string(resultalgebra$Lisp)
  algebra := formatMessages(algebra)
  -- At this point mathml contains the mathml for the output but does not
  -- include step number or type information.
  -- We should also save the command.
  -- I get the type and step number from the $internalHistoryTable
  axans:String := _
```
getInterp

The getInterp function is invoked when the HTTP request is a POST and contains the string "command". Essentially the game here is to rebind the various output streams used by Axiom so we can capture the normal output. This function returns a set of HTML 5 div blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

```lisp
getInterp(s:SExpression,command:String):Void ==
    WriteLine$Lisp concat ["getInterp: ",command]
    SETQ(tmpmathml$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
    SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
    ans := string parseAndEvalToStringEqNum$Lisp command
    SETQ(resultmathml$Lisp,_
        GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)$Lisp
    SETQ(resultalgebra$Lisp,_
        GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
```
SETQ($_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
SETQ($_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
CLOSE(tmpmathml$Lisp)$Lisp
CLOSE(tmpalgebra$Lisp)$Lisp
-- Since strings returned from axiom are going to be displayed in html I
-- should really check for the characters &,<,> and replace them with
-- &amp;&lt;&amp;.
-- At present I only check for ampersands in formatMessages.
mathml:String := string(resultmathml$Lisp)
algebra:String := string(resultalgebra$Lisp)
algebra := formatMessages(algebra)
-- At this point mathml contains the mathml for the output but does not
-- include step number or type information.
-- We should also save the command.
-- I get the type and step number from the $internalHistoryTable
axans:String := _
    concat ["<div class=_'stepnum_'>", lastStep(), "</div>
    <div class=_'command_'>", command, "</div>
    <div class=_'algebra_'>",algebra,"</div>
    <div class=_'mathml_'>",mathml,"</div>
    <div class=_'type_'>",lastType(),"</div>"]
-- WriteLine$Lisp concat ["mathml answer: ",mathml]
-- WriteLine$Lisp concat ["algebra answer: ",algebra]
q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
SiCopyStream(q,s)$Lisp
CLOSE(q)$Lisp
CLOSE(s)$Lisp

getLisp

The getLisp function is invoked when the HTTP request is a POST and contains the string
"lispcall".

| package AXSERV AxiomServer |

getLisp(s:SExpression,command:String):Void ==
    WriteLine$Lisp concat ["getLisp: ",command]
evalresult:=EVAL(READ_-FROM_-STRING(command)$Lisp)$Lisp
    mathml:STRING:=string(evalresult)
-- WriteLine$Lisp concat ["getLisp: after ",mathml]
-- WriteLine$Lisp concat ["getLisp output: ",mathml]
SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
SETQ(savemathml$Lisp, _$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
SETQ(resultalgebra$Lisp,_)
getShow

The getShow function is invoked when the HTTP request is a POST and contains the string "showcall". The )show command generates output to lisp's *standard-output* so we wrap that stream to capture it. The resulting string needs to be transformed into html-friendly form. This is done in the call to replace-entitites (see http.lisp)

— package AXSERV AxiomServer —

getShow(s:SExpression, showarg: String): Void ==
  WriteLine$Lisp concat ["getShow: ", showarg]
  realarg:= SUBSEQ(showarg, 6) $Lisp
  show:=
    "(progn (setq $options| '((|operations|))) (|show| |" realarg ")")"
  -- WriteLine$Lisp concat ["getShow: ", show]
  SETQ(SAVESTREAM$Lisp, _*STANDARD_-OUTPUT_*$Lisp)$Lisp
  SETQ(_*STANDARD_-OUTPUT_*$Lisp, _
    MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
  evalresult:= EVAL(READ_-FROM_-STRING(show)$Lisp)$Lisp
  SETQ(evalresult, _
GET-OUTPUT-STREAM-STRING(*STANDARD-OUTPUT-*Lisp)Lisp
SETQ(*STANDARD-OUTPUT-*Lisp,SAVESTREAM)Lisp
mathml:String:=string(REPLACE-ENTITIES(evalresult)Lisp)
SETQ(tmpalgebra:Lisp,MAKE-STRING-OUTPUT-STREAM())Lisp
SETQ(savemathml:Lisp,_,_texOutputStream)Lisp
SETQ(savealgebra:Lisp,_,_algebraOutputStream)Lisp
SETQ(_texOutputStream:Lisp,tmpmathml:Lisp)
SETQ(_algebraOutputStream:Lisp,tmpalgebra:Lisp)
SETQ(resultalgebra:Lisp,
GET-OUTPUT-STREAM-STRING(_algebraOutputStream:Lisp)Lisp)
SETQ(_texOutputStream:Lisp,savemathml:Lisp)
SETQ(_algebraOutputStream:Lisp,savealgebra:Lisp)
CLOSE(tmpalgebra:Lisp)
-- Since strings returned from axiom are going to be displayed in html I
-- should really check for the characters &,<,> and replace them with
-- &amp;,&lt;,&gt;.
-- At present I only check for ampersands in formatMessages.
algebra:String := string(resultalgebra:Lisp)
algebra := formatMessages(algebra)
-- At this point mathml contains the mathml for the output but does not
-- include step number or type information.
-- We should also save the command.
-- I get the type and step number from the $internalHistoryTable
axans:String :=
concat ["<div class="stepnum">", lastStep(), "]</div>_
            <div class="command">", showarg, "</div>_
            <div class="algebra">", algebra,"</div>_
            <div class="mathml">", mathml,"</div>_
            <div class="type">", lastType(), "</div>_
            ]
-- WriteLine$Lisp concat ["mathml answer: ", mathml]
q:=MAKE-STRING-INPUT-STREAM(axans)Lisp
SiCopyStream(q,s)Lisp
CLOSE(q)Lisp
CLOSE(s)Lisp

---

lastType
To examine the $internalHistoryTable use the following line

)lisp |$internalHistoryTable|

We need to pick out first member of internalHistoryTable and then pick out the element
with % as first element. Here is an example showing just the first element of the list, which
corresponds to the last command.

Note that the last command does not necessarily correspond to the last element of the first
element of $internalHistoryTable as it is in this example.
We also need to check for input error in which case the $internalHistoryTable is not changed and the type retrieved would be that for the last correct input.

---

```lisp
lastType():String ==
  SETQ(first$Lisp,FIRST(_$internalHistoryTable$Lisp)$Lisp)$Lisp
  count:Integer := 0
  hisLength:Integer := LIST_-LENGTH(_$internalHistoryTable$Lisp)$Lisp
  length:Integer := LIST_-LENGTH(first$Lisp)$Lisp
  -- This initializes stepSav. The test is a bit of a hack, maybe I’ll
  -- figure out the right way to do it later.
  if string stepSav$Lisp = "#<OBJNULL>" then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
  -- If hisLength = 0 then the history table has been reset to NIL
  -- and we're starting numbering over
  if hisLength = 0 then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
  if hisLength > 0 and
    car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp ^= stepSav$Lisp then
      SETQ(stepSav$Lisp,car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp)$Lisp
      while count < length repeat
        position(char "%",string FIRST(first$Lisp)$Lisp) = 2 =>
          count := length+1
        count := count +1
        SETQ(first$Lisp,REST(first$Lisp)$Lisp)$Lisp
      count = length + 1 =>
        string SECOND(SECOND(FIRST(first$Lisp)$Lisp)$Lisp)$Lisp

lastStep():String ==
  string car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp

formatMessages(str:String):String ==
  -- WriteLine("formatMessages")$Lisp
  -- I need to replace any ampersands with & and may also need to
  -- replace < and > with &lt; and &gt;
  strlist:List String
  WriteLine(str)$Lisp
  strlist := split(str,char "&")
  str := ""
  -- oops, if & is the last character in the string this method
  -- will eliminate it. Need to redo this.
```
for s in strlist repeat

   str := concat [str,s,"&"]
strlen:Integer := #str
str := str.(1..(#str - 5))
-- WriteLine(str)$Lisp
-- Here I split the string into lines and put each line in a "div".
strlist := split(str, char string NewlineChar$Lisp)
str := ""
-- WriteLine("formatMessages1")$Lisp
-- WriteLine(concat strlist)$Lisp
for s in strlist repeat

   WriteLine(s)$Lisp
   str := concat [str,"<div>",s,"</div>" ]

str

getContentType(pathvar:String):String ==
-- WriteLine("getContentType begin")$Lisp
-- set default content type
contentType:String := "text/plain"
-- need to test for successful match?
StringMatch(".*\.(.*)$", pathvar)$Lisp
u:UniversalSegment(Integer)
u := segment(MatchBeginning(1)$Lisp+1,_,
   MatchEnd(1)$Lisp)$UniversalSegment(Integer)
extension:String := pathvar.u
-- WriteLine$Lisp concat 
   ["file extension: ",extension]
-- test for extensions: html, htm, xml, xhtml, js, css
if extension = "html" then
   contentType:String := "text/html"
else if extension = "htm" then
   contentType:String := "text/html"
else if extension = "xml" then
   contentType:String := "text/xml"
else if extension = "xhtml" then
   contentType:String := "application/xhtml+xml"
else if extension = "js" then
   contentType:String := "text/javascript"
else if extension = "css" then
   contentType:String := "text/css"
else if extension = "png" then
   contentType:String := "image/png"
else if extension = "jpg" then
   contentType:String := "image/jpeg"
else if extension = "jpeg" then
   contentType:String := "image/jpeg"
-- WriteLine$Lisp concat ["Content-Type: ",contentType]
-- WriteLine("getContentType end")$Lisp
contentType
Chapter 3

Chapter B

package BALFACT BalancedFactorisation

— BalancedFactorisation.input —

)set break resume
)sys rm -f BalancedFactorisation.output
)spool BalancedFactorisation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BalancedFactorisation
--E 1

)spool
)lisp (bye)

— BalancedFactorisation.help —

BalancedFactorisation examples

This package provides balanced factorisations of polynomials.

See Also:
0 )show BalancedFactorisation
BalancedFactorisation (BALFACT)

Exports:
balancedFactorisation

— package BALFACT BalancedFactorisation —

)abbrev package BALFACT BalancedFactorisation
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 11 October 1991
++ Description:
++ This package provides balanced factorisations of polynomials.

BalancedFactorisation(R, UP): Exports == Implementation where
R : Join(GcdDomain, CharacteristicZero)
UP : UnivariatePolynomialCategory R

Exports ==> with
balancedFactorisation: (UP, UP) -> Factored UP
++ balancedFactorisation(a, b) returns
++ a factorisation \spad{a = p1^e1 ... pm^em} such that each
++ \spad{pi} is balanced with respect to b.
balancedFactorisation: (UP, List UP) -> Factored UP
++ balancedFactorisation(a, [b1,...,bn]) returns
++ a factorisation \spad{a = p1^e1 ... pm^em} such that each
++ pi is balanced with respect to \spad{[b1,...,bn]}.

Implementation ==> add
balSqr : (UP, Integer, List UP) -> Factored UP
balSqfr1: (UP, Integer, UP) -> Factored UP

balancedFactorisation(a:UP, b:UP) == balancedFactorisation(a, [b])

balSqfr1(a, n, b) ==
g := gcd(a, b)
fa := sqfrFactor((a exquo g)::UP, n)
ground? g => fa
fa * balSqfr1(g, n, (b exquo (g ** order(b, g))::UP))

balSqfr(a, n, l) ==
b := first l
empty? rest l => balSqfr1(a, n, b)
*/[balSqfr1(f.factor, n, b) for f in factors balSqfr(a,n,rest l)]

balancedFactorisation(a:UP, l:List UP) ==
empty?(ll := select(z1 +-> z1 ^= 0, l)) =>
  error "balancedFactorisation: 2nd argument is empty or all 0"
sa := squareFree a
unit(sa) * */[balSqfr(f.factor,f.exponent,ll) for f in factors sa]

package BOP1 BasicOperatorFunctions1

— BasicOperatorFunctions1.input —

)set break resume
/sys rm -f BasicOperatorFunctions1.output
/spool BasicOperatorFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BasicOperatorFunctions1
This package exports functions to set some commonly used properties of operators, including properties which contain functions.

See Also:
  o )show BasicOperatorFunctions1
++ Description:
++ This package exports functions to set some commonly used properties
++ of operators, including properties which contain functions.

BasicOperatorFunctions1(A:SetCategory): Exports == Implementation where
OP ==> BasicOperator
EVAL ==> "%eval"
CONST ==> "%constant"
DIFF ==> "%diff"
OUT ==> OutputForm
IN ==> InputForm

Exports ==> with
  evaluate : (OP, List A) -> Union(A, "failed")
  ++ evaluate(op, [a1,...,an]) checks if op has an "%eval"
  ++ property f. If it has, then \spad{f(a1,...,an)} is returned, and
  ++ "failed" otherwise.
  evaluate : (OP, List A -> A) -> OP
  ++ evaluate(op, foo) attaches foo as the "%eval" property
  ++ of op. If op has an "%eval" property f, then applying op
  ++ to \spad{(a1,...,an)} returns the result of \spad{f(a1,...,an)}.
  evaluate : (OP, A -> A) -> OP
  ++ evaluate(op, foo) attaches foo as the "%eval" property
  ++ of op. If op has an "%eval" property f, then applying op
  ++ to a returns the result of \spad{f(a)}. Argument op must be unary.
  evaluate : OP -> Union(List A -> A, "failed")
  ++ evaluate(op) returns the value of the "%eval" property of
  ++ op if it has one, and "failed" otherwise.
  derivative : (OP, List (List A -> A)) -> OP
  ++ derivative(op, [foo1,...,foon]) attaches [foo1,...,foon] as
  ++ the "%diff" property of op. If op has an "%diff" property
  ++ \spad{f1,...,fn)} then applying a derivation D
  ++ to \spad{op(a1,...,an)}
  ++ returns \spad{f1(a1,...,an) * D(a1) + ... + fn(a1,...,an) * D(an)}.
  derivative : (OP, A -> A) -> OP
  ++ derivative(op, foo) attaches foo as the "%diff" property
  ++ of op. If op has an "%diff" property f, then applying a
  ++ derivation D to op(a) returns \spad{f(a) * D(a)}.
  ++ Argument op must be unary.
  derivative : OP -> Union(List(List A -> A), "failed")
  ++ derivative(op) returns the value of the "%diff" property of
  ++ op if it has one, and "failed" otherwise.
if A has OrderedSet then
  constantOperator: A -> OP
  ++ constantOperator(a) returns a nullary operator op
  ++ such that \spad{op()} always evaluate to \spad{a}.
  constantOpIfCan : OP -> Union(A, "failed")
  ++ constantOpIfCan(op) returns \spad{a} if op is the constant
  ++ nullary operator always returning \spad{a}, "failed" otherwise.
Implementation => add
evaluate(op:OP, func:A -> A) ==
    evaluate(op, (ll:List(A)):A +-> func first ll)

evaluate op ==
    (func := property(op, EVAL)) case "failed" => "failed"
    (func::None) pretend (List A -> A)

evaluate(op:OP, args:List A) ==
    (func := property(op, EVAL)) case "failed" => "failed"
    ((func::None) pretend (List A -> A)) args

evaluate(op:OP, func:List A -> A) ==
    setProperty(op, EVAL, func pretend None)

derivative op ==
    (func := property(op, DIFF)) case "failed" => "failed"
    ((func::None) pretend List(List A -> A))

derivative(op:OP, grad:List(List A -> A)) ==
    setProperty(op, DIFF, grad pretend None)

derivative(op:OP, f:A -> A) ==
    unary? op or nary? op =>
        derivative(op, [(ll:List(A)):A +-> f first ll]$List(List A -> A))
    error "Operator is not unary"

if A has OrderedSet then
    cdisp : (OUT, List OUT) -> OUT
    csex : (IN, List IN) -> IN
    eqconst? : (OP, OP) -> Boolean
    ltconst? : (OP, OP) -> Boolean
    constOp : A -> OP
    opconst:OP :=
        comparison(equality(operator("constant"::Symbol, 0), eqconst?),
                   ltconst?)

cdisp(a, l) == a
    csex(a, l) == a
    eqconst?(a, b) ==
        (va := property(a, CONST)) case "failed" => not has?(b, CONST)
        ((vb := property(b, CONST)) case None) and
            ((va::None) pretend A) = ((vb::None) pretend A)
    ltconst?(a, b) ==
        (va := property(a, CONST)) case "failed" => has?(b, CONST)
        ((vb := property(b, CONST)) case None) and
            ((va::None) pretend A) < ((vb::None) pretend A)
constOp \ a \ == \\
\hspace{1em} \text{setProperty(} \\
\hspace{2em} \text{display(copy opconst, (ll:List(OUT)):OUT} \leftrightarrow \text{cdisp(a::OUT, ll)),} \\
\hspace{2em} \text{CONST, a pretend None)}

constantOpIfCan \ op \ == \\
\hspace{1em} \text{is?(op, "constant"::Symbol) and} \\
\hspace{2em} \left((u := \text{property(op, CONST)) case None} \right) \Rightarrow \left((u::\text{None}) \text{ pretend A} \right) "\text{failed}"

\text{if A has ConvertibleTo IN then} \\
\hspace{1em} \text{constantOperator a ==} \\
\hspace{2em} \text{input(constOp a, (ll:List(IN)):IN} \leftrightarrow \text{cssex(convert a, ll))} \\
\text{else} \\
\hspace{1em} \text{constantOperator a == constOp a}

— BOP1.dotabb —

"BOP1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BOP1"] \\
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"] \\
"BOP1" \rightarrow \text{"ALIST"}

— package BEZIER Bezier —

package \ BEZIER \ Bezier 

— Bezier.input —

)`set break resume` 
`sys rm -f Bezier.output` 
`spool Bezier.output` 
`set message test on` 
`set message auto off` 
`clear all` 
`--S 1 of 9` 
n: linearBezier([2.0,2.0],[4.0,4.0]) 
`--R` 
`--I (1) theMap(BEZIER;linearBezier;2LM;1!0,707)` 
`--R` 
`--E 1` 

Type: (Float -> List(Float))
[n(t/10.0) for t in 0..10 by 1]

---R (2)
---R [[2.0,2.0], [2.2,2.2], [2.4,2.4], [2.6,2.6], [2.8,2.8], [3.0,3.0],
---R [3.2,3.2], [3.4,3.4], [3.6,3.6], [3.8,3.8], [4.0,4.0]]
---R Type: List(List(Float))
---E 2

n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])

---I (3) theMap(BEZIER;quadraticBezier;3LM;2!0,291)
---R Type: (Float -> List(Float))
---E 3

[n(t/10.0) for t in 0..10 by 1]

---R (4)
---R [[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0],
---R [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]
---R Type: List(List(Float))
---E 4

n:=cubicBezier([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])

---I (5) theMap(BEZIER;cubicBezier;4LM;3!0,915)
---R Type: (Float -> List(Float))
---E 5

[n(t/10.0) for t in 0..10 by 1]

---R (6)
---R [[2.0,2.0], [2.112,2.54], [2.416,2.96], [2.864,3.26], [3.408,3.44],
---R [4.0,3.5], [4.592,3.44], [5.136,3.26], [5.584,2.96], [5.888,2.54],
---R [6.0,2.0]]
---R Type: List(List(Float))
---E 6

line:=[[i::Float,4.0] for i in -4..4 by 1]

---R (7)
---R [[- 4.0,4.0], [- 3.0,4.0], [- 2.0,4.0], [- 1.0,4.0], [0.0,4.0], [1.0,4.0],
---R [2.0,4.0], [3.0,4.0], [4.0,4.0]]
---R Type: List(List(Float))
functions:=[quadraticBezier([2.0,2.0],m,[6.0,2.0]) for m in line]

graphs:=[[point(((functions.i)(j/100.0))::LIST(DFLOAT)) _
    for j in 0..100] for i in 1..9]
CHAPTER 3. CHAPTER B

[2.375, 2.75], [2.4864000000000006, 2.7296000000000005],
[2.606, 2.7084000000000001], [2.7176, 2.6863999999999999],
[2.8374000000000006, 2.6636000000000006], [2.96, 2.6400000000000006],
[3.0859999999999999, 2.6160000000000006], [3.1903999999999999, 2.5863999999999999],
[3.2136000000000005, 2.5644], [3.3459999999999999, 2.5375999999999999],
[3.4784000000000005, 2.5099999999999998], [3.6150000000000002, 2.4815999999999998],
[3.7544, 2.4523999999999999], [3.8965999999999998, 2.4224000000000006],
[4.0415999999999999, 2.3915999999999999], [4.1894000000000009, 2.3604000000000006],
[4.3399999999999999, 2.3275999999999999], [4.4934000000000012, 2.2944],
[4.6496000000000013, 2.2604000000000006], [4.8086000000000002, 2.2256],
[4.9703999999999997, 2.1899999999999999], [5.1349999999999998, 2.1536],
[5.3023999999999996, 2.1164000000000005], [5.4725999999999999, 2.0783999999999998],
[5.6456, 2.0415999999999999], [5.8213999999999997, 2.0046000000000006],
[6.00, 2.00], [1.9212000000000002, 2.0396000000000001],
[1.8448, 2.0783999999999999], [1.7707999999999999, 2.1164000000000005],
[1.6920000000000003, 2.1536], [1.6299999999999999, 2.1899999999999999],
[1.5631999999999999, 2.2256], [1.4987999999999999, 2.2604000000000006],
[1.4368000000000003, 2.2944], [1.3772, 2.3275999999999999],
[1.3200000000000003, 2.3599999999999999], [1.2652000000000001, 2.3915999999999999],
[1.2128000000000001, 2.4224000000000006], [1.1628000000000003, 2.4523999999999999],
[1.0700000000000003, 2.5099999999999999], [1.0271999999999999, 2.5375999999999999],
[0.98680000000000012, 2.5644], [0.9487999999999998, 2.5903999999999999],
[0.9132000000000001, 2.6156000000000006], [0.8800000000000003, 2.6400000000000006],
[0.8491999999999995, 2.6636000000000006], [0.8207999999999997, 2.6863999999999999],
[0.7947999999999995, 2.7084000000000001], [0.7712, 2.7296000000000005],
[0.75, 2.75], [0.7311999999999996, 2.7696000000000005], [0.7147999999999999, 2.7884000000000002],
[0.6892000000000003, 2.8064], [0.6800000000000003, 2.8235999999999999],
[0.6732000000000003, 2.8419999999999999], [0.66799999999999995, 2.8636000000000006],
[0.66799999999999995, 2.8843999999999999], [0.6672000000000002, 2.9010000000000001],
[0.6752000000000002, 2.9216000000000006], [0.6827999999999996, 2.9323999999999999],
[0.6927999999999997, 2.9424000000000001], [0.7052000000000005, 2.9616],
[0.7199999999999997, 2.9676], [0.75680000000000014, 2.9744000000000002],
[0.77880000000000016, 2.9803999999999999],
\[\text{[0.80320000000000014,2.9855999999999998]},\]
\[\text{[0.82999999999999996,2.9900000000000002]},\]
\[\text{[0.85919999999999996,2.9935999999999998]}, \text{[0.89080000000000004,2.9964]},\]
\[\text{[1.13080000000000002,2.9964]}, \text{[1.17920000000000002,2.9935999999999998]},\]
\[\text{[1.23,2.9900000000000002]}, \text{[1.2831999999999999,2.9855999999999998]},\]
\[\text{[1.3388,2.9803999999999999]}, \text{[1.3968000000000003,2.9744000000000002]},\]
\[\text{[1.45720000000000003,2.9676]}, \text{[1.52,2.96]},\]
\[\text{[1.6528,2.9424000000000001]}, \text{[1.7228000000000003,2.9323999999999999]},\]
\[\text{[1.7951999999999999,2.9100000000000001]}, \text{[1.9472,2.8976000000000006]},\]
\[\text{[2.0268000000000006,2.8843999999999999]},\]
\[\text{[2.1080000000000005,2.8704000000000001]}, \text{[2.1932,2.8559999999999999]},\]
\[\text{[2.2799999999999998,2.8399999999999999]},\]
\[\text{[2.3692000000000002,2.8235999999999999]}, \text{[2.4607999999999999,2.8064]},\]
\[\text{[2.5480000000000002,2.7884000000000002]},\]
\[\text{[2.6511999999999999,2.76960000000000005]}, \text{[2.75,2.75]},\]
\[\text{[2.8512,2.72960000000000005]}, \text{[2.9548000000000005,2.70840000000000005]},\]
\[\text{[3.0608,2.6839999999999999]}, \text{[3.1692,2.66380000000000006]},\]
\[\text{[3.2799999999999999,2.6400000000000006]},\]
\[\text{[3.39320000000000002,2.6156000000000006]},\]
\[\text{[3.5087999999999999,2.5903999999999999]}, \text{[3.6267999999999998,2.5644]},\]
\[\text{[3.7471999999999999,2.5375999999999999]},\]
\[\text{[3.87000000000000001,2.5099999999999999]},\]
\[\text{[3.99520000000000005,2.4815999999999999]},\]
\[\text{[4.1229999999999999,2.4523999999999999]},\]
\[\text{[4.2579999999999997,2.42240000000000006]},\]
\[\text{[4.38520000000000001,2.3915999999999999]},\]
\[\text{[4.5199999999999999,2.3599999999999999]},\]
\[\text{[4.6571999999999999,2.3275999999999999]}, \text{[4.79680000000000011,2.2944]},\]
\[\text{[4.9387999999999996,2.26040000000000006]}, \text{[5.0831999999999997,2.2256]},\]
\[\text{[5.23000000000000004,2.1899999999999999]}, \text{[5.3792,2.1536]},\]
\[\text{[5.53080000000000001,2.11640000000000006]},\]
\[\text{[5.68480000000000001,2.0783999999999998]},\]
\[\text{[5.8411999999999997,2.0396000000000001]}, \text{[6.,2.]}\]
[5.8609999999999999, 2.0396000000000001], [6.2, 2.]

[2.2, 2.], [1.9608000000000003, 2.0396000000000001],

[1.9232, 2.0783999999999999], [1.8872, 2.1164000000000005],

[1.7887999999999999, 2.2256], [1.7592000000000003, 2.2604000000000006],

[1.7312000000000003, 2.2944], [1.7048000000000001, 2.3276000000000006],

[1.6799999999999999, 2.3599999999999999],

[1.6568000000000001, 2.3916000000000002], [1.6352, 2.4244000000000006],

[1.6152, 2.4524000000000006], [1.5968, 2.4815999999999999],

[1.5800000000000001, 2.5099999999999999], [1.5648, 2.5375999999999999],

[1.5519999999999999, 2.5644], [1.5391999999999999, 2.5903999999999999],

[1.5287999999999999, 2.6156000000000006], [1.52, 2.6400000000000006],

[1.5127999999999999, 2.6636000000000006],

[1.5072000000000001, 2.6863999999999999], [1.5032000000000001, 2.7084000000000006],

[1.5008000000000001, 2.7296000000000005], [1.5, 2.75],

[1.5008000000000001, 2.7696000000000005],

[1.5032000000000001, 2.7884000000000006], [1.5072000000000001, 2.8064],

[1.5127999999999999, 2.8236000000000002], [1.52, 2.8400000000000006],

[1.5287999999999999, 2.8556000000000006], [1.5391999999999999, 2.8704000000000001],

[1.5519999999999999, 2.8843999999999999], [1.5648, 2.8976000000000006],

[1.5800000000000001, 2.9100000000000002], [1.5968, 2.9216000000000006],

[1.6152, 2.9324000000000006], [1.6352, 2.9424000000000001],

[1.6568000000000001, 2.96], [1.6799999999999999, 2.9803999999999999],

[1.7168000000000001, 2.9903999999999999], [1.7592000000000003, 2.9855999999999999],

[1.8200000000000003, 2.9900000000000006],

[1.8528000000000002, 2.9935999999999999], [1.8872, 2.9964],

[1.9232, 2.9984000000000002], [1.9608000000000003, 2.9996], [2.3],

[2.0479999999999999, 2.9996], [2.0832000000000006], [2.9984000000000002],

[2.1272000000000002], [2.1728000000000005], [2.9935999999999999],

[2.2200000000000006], [2.9900000000000002],

[2.2688000000000006], [2.9855999999999999],

[2.3191999999999998], [2.9803999999999999], [2.3712, 2.9744000000000002],

[2.4247999999999999], [2.9672], [2.5367999999999999], [2.9516],

[2.5952000000000002], [2.4240000000000006], [2.9424000000000001],

[2.6519999999999998], [2.9329999999999999],

[2.7168000000000001], [2.9216000000000006],

[2.7799999999999998], [2.9100000000000001],

[2.8448000000000002], [2.8976000000000006], [2.9112, 2.8843999999999999],

[2.9792000000000005], [2.8704000000000001], [3.0488, 2.8555999999999999],

[3.1200000000000001], [2.8399999999999999],

[3.1926000000000001], [2.8235999999999999], [3.2671999999999999], [2.8064],

[3.3419999999999999], [2.7840000000000002],

[3.4207999999999999], [2.7696000000000006], [3.5, 2.75],

[3.5808, 2.7296000000000005], [3.6631999999999998], [2.7084000000000001],
\begin{itemize}
\item \([3.7471999999999999, 2.6863999999999999, 2.6156000000000006], \]
\item \([3.8328000000000007, 2.6636000000000006], \]
\item \([3.9199999999999999, 2.6400000000000006], \]
\item \([4.0087999999999999, 2.6156000000000006], \]
\item \([4.0991999999999999, 2.5903999999999998], \]
\item \([4.1912000000000003, 2.5644], \]
\item \([4.2847999999999997, 2.5375999999999999], \]
\item \([4.3799999999999999, 2.5099999999999998], \]
\item \([4.4767999999999999, 2.4815999999999998], \]
\item \([4.5751999999999997, 2.4523999999999999], \]
\item \([4.6752000000000002, 2.4224000000000006], \]
\item \([4.7767999999999997, 2.3915999999999999], \]
\item \([4.8799999999999999, 2.3599999999999999], \]
\item \([4.9847999999999999, 2.3275999999999999], \]
\item \([5.0911999999999997, 2.2944], \]
\item \([5.1992000000000012, 2.2604000000000006], \]
\item \([5.3087999999999997, 2.2256], \]
\item \([5.4199999999999999, 2.1899999999999999], \]
\item \([5.5327999999999999, 2.1536], \]
\item \([5.6471999999999998, 2.1164000000000005], \]
\item \([5.76320000000000012, 2.0783999999999998], \]
\item \([5.8807999999999998, 2.0396000000000001], \]
\item \([6., 2.] \]
\item \([1.9805999999999999, 2.0396000000000001], \]
\item \([1.9623999999999999, 2.0783999999999998], \]
\item \([1.9454000000000002, 2.1164000000000005], \]
\item \([1.915, 2.1899999999999999], \]
\item \([1.8784000000000001, 2.2944], \]
\item \([1.8600000000000003, 2.3599999999999999], \]
\item \([1.8526000000000002, 2.3915999999999999], \]
\item \([1.8413999999999999, 2.4523999999999999], \]
\item \([1.8375999999999999, 2.4815999999999998], \]
\item \([1.8335999999999999, 2.5375999999999999], \]
\item \([1.8344, 2.5803999999999999], \]
\item \([1.8400000000000003, 2.6400000000000006], \]
\item \([1.8446000000000002, 2.6636000000000006], \]
\item \([1.8504, 2.6863999999999999], \]
\item \([1.8579999999999999, 2.7084000000000001], \]
\item \([1.8655999999999999, 2.7296000000000005], \]
\item \([1.875, 2.75], \]
\item \([1.8855999999999999, 2.7690000000000005], \]
\item \([1.8974, 2.7884000000000002], \]
\item \([1.9040000000000001, 2.8064], \]
\item \([1.9246000000000003, 2.8235999999999999], \]
\item \([1.9399999999999999, 2.8399999999999999], \]
\item \([1.9565999999999999, 2.8559999999999999], \]
\item \([1.9743999999999999, 2.8704000000000001], \]
\item \([1.9934000000000003, 2.8843999999999999], \]
\item \([2.0135999999999999, 2.8976000000000006], \]
\item \([2.0350000000000001, 2.9100000000000006], \]
\item \([2.0575999999999999, 2.9216000000000006], \]
\item \([2.0813999999999999, 2.9323999999999999], \]
\item \([2.1063999999999999, 2.9424000000000001], \]
\item \([2.1326000000000001, 2.9516], \]
\item \([2.1600000000000001, 2.96], \]
\item \([2.1839999999999999, 2.9744000000000002], \]
\item \([2.2494000000000005, 2.9803999999999999], \]
\item \([2.2816000000000001, 2.9855999999999999], \]
\end{itemize}
---R [2.3149999999999999, 2.9900000000000002],
---R [2.3496000000000006, 2.9935999999999998], [2.3854000000000006, 2.9964],
---R [2.4224000000000006, 2.9984000000000002], [2.4605999999999999, 2.9996],
---R [2.4224000000000006, 2.9984000000000002], [2.4605999999999999, 2.9996],
---R [2.5,3.], [2.5406, 2.9996], [2.5823999999999998, 2.9984000000000002],
---R [2.6254, 2.9964], [2.6696, 2.9935999999999998],
---R [2.7149999999999999, 2.9900000000000002],
---R [2.7616000000000005, 2.9935999999999998], [2.8094000000000001, 2.9903999999999999],
---R [2.8584000000000005, 2.9964], [2.9085999999999999, 2.9996], [2.96, 2.96],
---R [3.0663999999999998, 2.9424000000000001], [3.1214, 2.9323999999999999],
---R [3.1776, 2.9216000000000006], [3.2349999999999999, 2.9100000000000001],
---R [3.2936000000000005, 2.8976000000000006], [3.3534000000000006, 2.8843999999999999],
---R [3.4144000000000005, 2.8704000000000006], [3.4765999999999999, 2.8555999999999999],
---R [3.54, 2.8399999999999999], [3.6046000000000005, 2.8235999999999999],
---R [3.6734000000000001, 2.7884000000000002], [3.8094000000000001, 2.7696000000000005],
---R [3.9465999999999998, 2.7296000000000006], [4.0174000000000003, 2.7084000000000001],
---R [4.0903999999999998, 2.6863999999999999], [4.1646000000000001, 2.6636000000000006],
---R [4.2400000000000002, 2.6400000000000006], [4.3166000000000001, 2.6156000000000006],
---R [4.3944000000000001, 2.5903999999999998], [4.4733999999999998, 2.5644],
---R [4.55360000000000012, 2.5375999999999999], [4.6349999999999998, 2.5099999999999999],
---R [4.71760000000000009, 2.4815999999999999], [4.8014000000000001, 2.4523999999999999],
---R [4.8840000000000001, 2.4224000000000006], [4.7258999999999999, 2.3915999999999999],
---R [5.0599999999999999, 2.3599999999999999], [5.1486000000000001, 2.3275999999999998],
---R [5.2384000000000004, 2.2944], [5.3293999999999999, 2.2604000000000006],
---R [5.4215999999999998, 2.2256], [5.5149999999999997, 2.1899999999999999],
---R [5.6096000000000004, 2.1536], [5.7054000000000009, 2.1164000000000005],
---R [5.8023999999999996, 2.0783999999999999], [5.9005999999999998, 2.0396000000000001],
---R [6.0140000000000001, 2.0004, 2.0396000000000001], [6.2.] }
<table>
<thead>
<tr>
<th>x0</th>
<th>y0</th>
<th>x1</th>
<th>y1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2.08</td>
<td>2.0396</td>
</tr>
<tr>
<td>2.04</td>
<td>2.0396</td>
<td>2.12</td>
<td>2.1164</td>
</tr>
<tr>
<td>2.2</td>
<td>2.2256</td>
<td>2.27</td>
<td>2.2604</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2944</td>
<td>2.35</td>
<td>2.3275</td>
</tr>
<tr>
<td>2.39</td>
<td>2.2944</td>
<td>2.43</td>
<td>2.3199</td>
</tr>
<tr>
<td>2.48</td>
<td>2.2944</td>
<td>2.52</td>
<td>2.2640</td>
</tr>
<tr>
<td>2.56</td>
<td>2.2944</td>
<td>2.60</td>
<td>2.2420</td>
</tr>
<tr>
<td>2.64</td>
<td>2.2944</td>
<td>2.72</td>
<td>2.2000</td>
</tr>
<tr>
<td>2.76</td>
<td>2.2944</td>
<td>2.83</td>
<td>2.1640</td>
</tr>
<tr>
<td>2.87</td>
<td>2.2944</td>
<td>2.92</td>
<td>2.1164</td>
</tr>
<tr>
<td>2.96</td>
<td>2.2944</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

---

CHAPTER 3. CHAPTER B
--R [5.6399999999999997, 2.3275999999999999], [5.6799999999999997, 2.2944000000000006],
--R [5.7199999999999998, 2.2604000000000006], [5.7599999999999998, 2.2256000000000006],
--R [5.7999999999999998, 2.1899999999999999], [5.8399999999999999, 2.1536000000000005],
--R [5.8799999999999999, 2.1164000000000005],
--R [5.9199999999999999, 2.0783999999999998], [5.96, 2.0396000000000001],
--R [6., 2.]]
--R ]
--R Type: List(List(Point(DoubleFloat)))
--E 9

-- We do not do these during testing since graphics is not available
-- The resulting image is in the Bezier section of Volume 10.4

--d1:=draw(graphs.1,title="Bezier Control Point Motion")
--others:=[graphs.i for i in 2..9]
--for i in 2..9 for graph in others repeat putGraph(d1,[graph],i)
--vp:=makeViewport2D(d1)

BezoutMatrix contains functions for computing resultants and discriminants using Bezout matrices.
A linear Bezier curve is a simple interpolation between the starting point and the ending point based on a parameter t.

Given a start point $a=[x_1,y_1]$ and an endpoint $b=[x_2,y_2]$
\[
f(t) = [(1-t)x_1 + tx_2, (1-t)y_1 + ty_2]
\]
\[
n:=\text{linearBezier}([2.0,2.0],[4.0,4.0])
\]
\[
\text{theMap(BEZIER;linearBezier;2LM;110,707)}
\]
\[
[n(t/10.0) \text{ for } t \text{ in } 0..10 \text{ by } 1]
\]
\[
[[2.0,2.0], [2.2,2.2], [2.4,2.4], [2.6,2.6], [2.8,2.8], [3.0,3.0], [3.2,3.2], [3.4,3.4], [3.6,3.6], [3.8,3.8], [4.0,4.0]]
\]

A quadratic Bezier curve is a simple interpolation between the starting point, a middle point, and the ending point based on a parameter t.

Given a start point $a=[x_1,y_1]$, a middle point $b=[x_2,y_2]$, and an endpoint $c=[x_3,y_3]$
\[
f(t) = [(1-t)^2 x_1 + 2t(1-t) x_2 + t^2 x_3, (1-t)^2 y_1 + 2t(1-t) y_2 + t^2 y_3]
\]
\[
n:=\text{quadraticBezier}([2.0,2.0],[4.0,4.0],[6.0,2.0])
\]
\[
\text{theMap(BEZIER;quadraticBezier;3LM;210,291)}
\]
\[
[n(t/10.0) \text{ for } t \text{ in } 0..10 \text{ by } 1]
\]
\[
[[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0], [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]
\]

A cubic Bezier curve is a simple interpolation between the starting point, a left-middle point, a right-middle point, and the ending point based on a parameter t.

Given a start point $a=[x_1,y_1]$, the left-middle point $b=[x_2,y_2]$, the right-middle point $c=[x_3,y_3]$ and an endpoint $d=[x_4,y_4]$
\[
f(t) = [(1-t)^3 x_1 + 3t(1-t)^2 x_2 + 3t^2 (1-t) x_3 + t^3 x_4, (1-t)^3 y_1 + 3t(1-t)^2 y_2 + 3t^2 (1-t) y_3 + t^3 y_4]
\]
\[
n:=\text{cubicBezier}([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])
\]
\[
\text{theMap(BEZIER;cubicBezier;4LM;310,915)}
\]
\[
[n(t/10.0) \text{ for } t \text{ in } 0..10 \text{ by } 1]
\]
\[
[[2.0,2.0], [2.112,2.54], [2.416,2.96], [2.816,3.26], [3.408,3.44], [4.0,3.5], [4.592,3.44], [5.136,3.26], [5.584,2.96], [5.888,2.54], [6.0,2.0]]
\]

Bezier curves "move" based on moving their control points, which
in the case of the three components of a quadratic Bezier curve are the three points given. To see this motion we can show what happens when you "drag" the middle control point along the line from \([-4,4]\) to \([4,4]\) by increments of 1.

First, we form the line as a list of Floats, 9 in all.

\[
\text{line:}=[\{i::\text{Float}, 4.0\} \text{ for } i \in \{\ldots -4 \ldots 4\} \text{ by } 1]
\]

Next, we form a list of functions, each with a different center control point. Notice that the endpoints remain fixed so we expect that the curve will begin and end at the same point but that the midpoint is pulled around.

\[
\text{functions:}=[\text{quadraticBezier}(\{2.0,2.0\}, m, \{6.0,2.0\}) \text{ for } m \in \text{line}]
\]

Then we form a list of the graphs by calling each function in the above list. Each function call happens 101 times (to include both endpoints). Thus we get a List of Lists of Points of DoubleFloats

\[
\text{graphs:}=[\{\text{point}\left(\left(\left(\text{functions}.i\right)(j/100.0)\right):\text{LIST}(\text{DFLOAT})\right)\}_j\text{ for } j \in \{0 \ldots 100\} \text{ for } i \in \{1 \ldots 9\}]
\]

We draw the first graph to see if it looks reasonable:

\[
\text{d1:=draw(graphs}.1)
\]

Since it does we add the other 8 graphs. The 2D graphs can hold up to 9 simultaneous graphs.

\[
\text{others:=}=[\text{graphs}.i \text{ for } i \in \{2 \ldots 9\}]
\] for \(i \in \{2 \ldots 9\}\) for graph in others repeat putGraph(d1, [graph], i)

and now we display them all on one graph.

\[
\text{vp:=makeViewport2D(d1)}
\]

See Also:
- )show Bezier
- )show TwoDimensionalViewport
- )d op draw
- )d op point
- )d op putGraph
- )d op makeViewport2D
Beziers (BEZIER)

Exports:
linearBezier quadraticBezier cubicBezier

— package BEZIER Bezier —

)abbrev package BEZIER Bezier
++ Author: Timothy Daly
++ Date Created: 14 April 2009
++ Description:
++ Provide linear, quadratic, and cubic spline bezier curves

Beziers(R:Ring): with
   linearBezier: (x:List R,y:List R) -> Mapping(List R,R)
      ++ A linear Bezier curve is a simple interpolation between the
      ++ starting point and the ending point based on a parameter t.
      ++ Given a start point a=[x1,y1] and an endpoint b=[x2,y2]
      ++ f(t) == [(1-t)*x1 + t*x2, (1-t)*y1 + t*y2]
      ++
      ++X n:linearBezier([2.0,2.0],[4.0,4.0])
      ++X [n(t/10.0) for t in 0..10 by 1]
   quadraticBezier: (x:List R,y:List R,z:List R) -> Mapping(List R,R)
      ++ A quadratic Bezier curve is a simple interpolation between the
      ++ starting point, a middle point, and the ending point based on
      ++ a parameter t.
      ++ Given a start point a=[x1,y1], a middle point b=[x2,y2],
      ++ and an endpoint c=[x3,y3]
      ++ f(t) == [(1-t)^2 x1 + 2t(1-t) x2 + t^2 x3,
      ++ (1-t)^2 y1 + 2t(1-t) y2 + t^2 y3]
      ++
      ++X n:quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
      ++X [n(t/10.0) for t in 0..10 by 1]
      ++ A cubic Bezier curve is a simple interpolation between the
      ++ starting point, a left-middle point, a right-middle point,
      ++ and the ending point based on a parameter t.
Given a start point \(a = [x_1, y_1]\), the left-middle point \(b = [x_2, y_2]\),
the right-middle point \(c = [x_3, y_3]\) and an endpoint \(d = [x_4, y_4]\)
\[
f(t) = [(1-t)^3 x_1 + 3t(1-t)^2 x_2 + 3t^2 (1-t) x_3 + t^3 x_4,
(1-t)^3 y_1 + 3t(1-t)^2 y_2 + 3t^2 (1-t) y_3 + t^3 y_4]
\]

\[X n:=\text{cubicBezier}([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])\]
\[X [n(t/10.0) for t in 0..10 by 1]\]

```
linearBezier(a,b) ==
t +-> [(1-t)*(a.1) + t*(b.1), (1-t)*(a.2) + t*(b.2)]

quadraticBezier(a,b,c) ==
t +-> [(1-t)**2*(a.1) + 2*t*(1-t)*(b.1) + t**2*(c.1),
      (1-t)**2*(a.2) + 2*t*(1-t)*(b.2) + t**2*(c.2)]

cubicBezier(a,b,c,d) ==
t +-> [(1-t)**3*(a.1) + 3*t*(1-t)**2*(b.1)
     + 3*t**2*(1-t)*(c.1) + t**3*(d.1),
      (1-t)**3*(a.2) + 3*t*(1-t)**2*(b.2)
     + 3*t**2*(1-t)*(c.2) + t**3*(d.2)]
```

—— BEZIER.dotabb ——

```
"BEZIER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BEZIER"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"BEZIER" -> "LMODULE"
"BEZIER" -> "SGROUP"
```

package BEZOUT BezoutMatrix

—— BezoutMatrix.input ——

```lisp
)set break resume
)sys rm -f BezoutMatrix.output
)spool BezoutMatrix.output
)set message test on
)set message auto off
)clear all
```
BezoutMatrix (BEZOUT)

Exports:
bezoutDiscriminant bezoutMatrix bezoutResultant sylvesterMatrix

— package BEZOUT BezoutMatrix —

)abbrev package BEZOUT BezoutMatrix
++ Author: Clifton J. Williamson
++ Date Created: 2 August 1988
++ Date Last Updated: 3 November 1993
++ Reference: Knuth, The Art of Computer Programming, 2nd edition,
++ Description:
++ \texttt{BezoutMatrix} contains functions for computing resultants and
++ discriminants using Bezout matrices.

\texttt{BezoutMatrix(R,UP,M,Row,Col): Exports == Implementation where}
R : Ring
UP : UnivariatePolynomialCategory R
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)
I ==> Integer
lc ==> leadingCoefficient

Exports ==> with
sylvesterMatrix: (UP,UP) -> M
++ sylvesterMatrix(p,q) returns the Sylvester matrix for the two
++ polynomials p and q.
bezoutMatrix: (UP,UP) -> M
++ bezoutMatrix(p,q) returns the Bezout matrix for the two
++ polynomials p and q.

if R has commutative("*") then
bezoutResultant: (UP,UP) -> R
++ bezoutResultant(p,q) computes the resultant of the two
++ polynomials p and q by computing the determinant of a Bezout matrix.

bezoutDiscriminant: UP -> R
++ bezoutDiscriminant(p) computes the discriminant of a polynomial p
++ by computing the determinant of a Bezout matrix.

Implementation ==> add

sylvesterMatrix(p,q) ==
n1 := degree p; n2 := degree q; n := n1 + n2
sylmat : M := new(n,n,0)
minR := minRowIndex sylmat; minC := minColIndex sylmat
maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
p0 := p
-- fill in coefficients of 'p'
while not zero? p0 repeat
  coef := lc p0; deg := degree p0; p0 := reductum p0
  -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
  for i in 0..n2 - 1 repeat
    qsetelt!(sylmat,minR + i,minC + n1 - deg + i,coef)
q0 := q
-- fill in coefficients of 'q'
while not zero? q0 repeat
    coef := lc q0; deg := degree q0; q0 := reductum q0
    for i in 0..n1-1 repeat
        qsetelt_!(sylmat,minR + n2 + i,minC + n2 - deg + i,coef)
sylmat

bezoutMatrix(p,q) ==
-- This function computes the Bezout matrix for 'p' and 'q'.
-- One must have deg(p) >= deg(q), so the arguments are reversed
-- if this is not the case.
    n1 := degree p; n2 := degree q; n := n1 + n2
    n1 < n2 => bezoutMatrix(q,p)
    m1 : I := n1 - 1; m2 : I := n2 - 1; m : I := n - 1
    -- 'sylmat' will be a matrix consisting of the first n1 columns
    -- of the standard Sylvester matrix for 'p' and 'q'
    sylmat : M := new(n,n1,0)
    minR := minRowIndex sylmat; minC := minColIndex sylmat
    maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
    p0 := p
    -- fill in coefficients of 'p'
    while not ground? p0 repeat
        coef := lc p0; deg := degree p0; p0 := reductum p0
        -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
        -- for i = 0...
        -- quit when i > m2 or when i + (n1 - k) > m1, whichever happens first
        for i in 0..min(m2,deg - 1) repeat
            qsetelt_!(sylmat,minR + i,minC + n1 - deg + i,coef)
    q0 := q
    -- fill in coefficients of 'q'
    while not zero? q0 repeat
        coef := lc q0; deg := degree q0; q0 := reductum q0
        -- put ak = coef(q,k) in sylmat(minR + n1 + i,minC + i + (n2 - k))
        -- for i = 0...
        -- quit when i > m1 or when i + (n2 - k) > m1, whichever happens first
        -- since n2 - k >= 0, we quit when i + (n2 - k) > m1
        for i in 0..(deg + n1 - n2 - 1) repeat
            qsetelt_!(sylmat,minR + n2 + i,minC + n1 - deg + i,coef)
    -- 'bezmat' will be the 'Bezout matrix' as described in Knuth
    bezmat : M := new(n1,n1,0)
    for i in 0..m2 repeat
        -- replace A_i by (b_0 A_i + ... + b_{n-2-i} A_{n-2-i}) A_{n-2-i}) -
        -- (a_0 B_i + ... + a_{n-2-i} B_{n-2-i}), as in Knuth
        bound : I := n2 - i; q0 := q
        while not zero? q0 repeat
            deg := degree q0
            if (deg < bound) then
                -- add b_deg A_{n-2} to the new A_i
                coef := lc q0
                for k in minC..maxC repeat


c := coef * qelt(sylmat, minR + m2 - i - deg, k) + qelt(bezmat, minR + m2 - i, k) qsetelt_!(bezmat, minR + m2 - i, k, c)
q0 := reductum q0
p0 := p
while not zero? p0 repeat
  deg := degree p0
  if deg < bound then
    coef := lc p0
    -- subtract a_deg B_{n_2 - deg} from the new A_i
    for k in minC..maxC repeat
      c := -coef * qelt(sylmat, minR + m - i - deg, k) + qelt(bezmat, minR + m2 - i, k)
      qsetelt_!(bezmat, minR + m2 - i, k, c)
p0 := reductum p0
for i in n2..m1 repeat for k in minC..maxC repeat
  qsetelt_!(bezmat, minR + i, k, qelt(sylmat, minR + i, k))
bezmat

if R has commutative("*") then

  bezoutResultant(f, g) == determinant bezoutMatrix(f, g)

if R has IntegralDomain then

  bezoutDiscriminant f ==
    degMod4 := (degree f) rem 4
    (degMod4 = 0) or (degMod4 = 1) =>
      (bezoutResultant(f, differentiate f) exquo (lc f)) :: R
    -((bezoutResultant(f, differentiate f) exquo (lc f)) :: R)
  else

    bezoutDiscriminant f ==
      lc f = 1 =>
        degMod4 := (degree f) rem 4
        (degMod4 = 0) or (degMod4 = 1) =>
          bezoutResultant(f, differentiate f)
          -bezoutResultant(f, differentiate f)
        error "bezoutDiscriminant: leading coefficient must be 1"

— BEZOUT.dotabb —

"BEZOUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BEZOUT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BEZOUT" -> "PFECAT"
package BLUPPACK BlowUpPackage

---

BlowUpPackage examples
---

The following is part of the PAFF package
See Also:
- \( \text{show \text{BlowUpPackage}} \)

---

**BlowUpPackage (BLUPPACK)**

```plaintext
---

\( \text{package BLUPPACK BlowUpPackage} \)---
```

---

```
)abbrev package BLUPPACK BlowUpPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
BlowUpPackage(K,symb,PolyRing,E, BLMET):Exports == Implementation where
  K : Field
  symb : List Symbol
  PolyRing : FiniteAbelianMonoidRing(K,E)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  BLMET : BlowUpMethodCategory

  NNI ==> NonNegativeInteger
  RFP ==> RootsFindingPackage
  NP ==> NewtonPolygon( K, BlUpRing, E2 , #bls)
  PackPoly ==> PackageForPoly(K,BlUpRing,E2,#bls)
  bls ==> ['X,'Y]
  BlUpRing ==> DistributedMultivariatePolynomial( bls , K )
  E2 ==> DirectProduct(#bls , NNI)
  AFP ==> AffinePlane(K)
```
PACKAGE BLUPPACK BLOWUPPACKAGE

blowUpRec ==> Record(recTransStr:BlUpRing,recPoint:AFP,recChart:BLMET, _
    definingExtension:K)
blowUpReturn ==> Record(mult:NonNegativeInteger,subMult: NNI, _
    blUpRec:List(blowUpRec))
recStr ==> Record( sM: NNI , blRec:List blowUpRec)

Exports ==> with

    applyTransform: (PolyRing,BLMET) -> PolyRing
    ++ quadTransform(pol,chart) apply the quadratique transformation to
    ++ pol specified by chart which consist of 3 integers. The last one
    ++ indicates which varibles is set to 1, the first on indicates
    ++ which variable remains unchange, and the second one indicates
    ++ which variable oon which the transformation is applied.
    ++ For example, [2,3,1] correspond to the following:
    ++ x -> 1, y -> y, z -> yz (here the variable are [x,y,z] in BlUpRing).
quadTransform: (BlUpRing,NNI,BLMET) -> BlUpRing -- CHH
    ++ quadTransform(pol,n,chart) apply the quadratique transformation
    ++ to pol specified by chart has in quadTransform(pol,chart) and
    ++ extract x**n to it, where x is the variable specified by the
    ++ first integer in chart (blow-up exceptional coordinate).
stepBlowUp:(BlUpRing,AFP,BLMET,K) -> blowUpReturn -- CHH
    ++ stepBlowUp(pol,pt,n) blow-up the point pt on the curve defined
    ++ by pol in the affine neighbourhood specified by n.
newtonPolySlope: BlUpRing -> List List(NNI)
polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing
biringToPolyRing: (BlUpRing, BLMET) -> PolyRing

Implementation ==> add

    import BlUpRing
    import AFP
    import RFP(K)
    import PackPoly
    import NP

    makeAff( l:List(K) , chart: BLMET ):AFP ==
        (excepCoord chart) = 1 => affinePoint( l )$AFP
        affinePoint( reverse l )$AFP

    blowExp: (E2, NNI, BLMET ) -> E2

    maxOf: (K,K) -> K
getStrTrans: ( BlUpRing , List BlUpRing , BLMET, K ) -> recStr

stepBlowUp(crb:BlUpRing,pt:AFP,chart:BLMET,actualExtension:K) ==
-- next is with Hamburger-Noether method
BLMET has HamburgerNoether =>
  nV:Integer:= chartCoord chart
crbTrans:BlUpRing:=translate(crb, list(pt))$PackPoly
newtPol:= newtonPolygon( crbTrans, quotValuation chart, _
    ramifMult chart, type chart )$NP
multPt:= multiplicity(newtPol)$NP
one?(multPt) =>
  [multPt, 0 , empty() ]$blowUpReturn
listOfgetTr:List recStr:=
    [ getStrTrans( crbTrans , edge , chart , actualExtension ) _
      for edge in newtPol ]
lsubM: List NNI := [ ll.sM for ll in listOfgetTr]
subM := reduce( "+", lsubM )
llistOfRec: List List blowUpRec := [ ll.blRec for ll in listOfgetTr]
listOfRec:= concat llistOfRec
  [ multPt, subM ,listOfRec]$blowUpReturn
-- next is with usual quadratic transform.

BLMET has QuadraticTransform =>
  nV:Integer:= chartCoord chart
  lpt:List(K) := list(pt)$AFP
crbTrans:=translate(crb,lpt)
minForm:=minimalForm(crbTrans)
multPt:=totalDegree( minForm)$PackPoly
listRec:List blowUpRec:=empty()
one?(multPt) => [multPt, 0 , listRec]$blowUpReturn
-- now pt is singular !!!
lstInd:=[i::PositiveInteger for i in 1..2 ]
-- la ligne suivante fait un choix judicieux pour minimiser le
-- degre' du transforme' stricte.
if degree( crbTrans , 2 )$PackPoly < degree( crbTrans , 1 )$PackPoly _
  then lstInd := reverse lstInd
ptInf:List(K)=[0$K,0$K]
laCarte:BLMET:=
  ([(last(lstInd), first(lstInd),nV) @ List Integer] @ BLMET
laCarteInf:BLMET:=
  ([first(lstInd),last(lstInd),nV] @ List Integer ) @ BLMET
transStricte :=quadTransform(crbTrans,multPt,laCarte)
transStricteInf:=quadTransform(crbTrans,multPt,laCarteInf)
listPtsSingEcl:List: AFP:=empty()
transStricteZero:BlUpRing:= replaceVarByOne(minForm,excepCoord laCarte)
recOfZeros:=
  distinguishedRootsOf(univariate(transStricteZero)$PackPoly ,_
    actualExtension )$RFP(K)
degExt:=recOfZeros.extDegree
"one?(degExt) =>
print(("You need an extension of degree")::OutputForm)
print(degExt::OutputForm)
error("Have a nice day")
listPtsSingEcl:=[makeAff([0$K,a]::List(K),laCarte) |
  for a in recOfZeros.zeros]
listRec:=[
  [ transStricte,_,
    ptS,laCarte,_,
    maxOf(a,actualExtension)]$blowUpRec_
  for ptS in listPtsSingEcl_
    for a in recOfZeros.zeros]
if zero?(constant(transStricteInf))$K then
  listRec:= concat(listRec,[transStricteInf,_,
    affinePoint(ptInf)$AFP,_,
    laCarteInf,_,
    actualExtension]$blowUpRec)
  empty?(listRec) =>
    error "Something is very wrong in blowing up!!!!!!"
  [multPt, 0 ,listRec]$blowUpReturn
error "Desingularisation is not implemented for the blowing up method chosen, see BlowingUpMethodCategory."

getStrTrans( crb , inedge , actChart, actualExtension ) ==
edge:= copy inedge
s := slope(edge)$NP
sden:Integer
snum:Integer
i1:Integer
i2:Integer
if s.type case "right" then
  sden:= s.base
  snum:=s.height
  i1:=1
  i2:=2
else -- interchange les roles de X et Y .
  sden:= s.height
  snum:= s.base
  i1:=2
  i2:=1
  edge := copy reverse inedge
ee := entries( degree first edge) pretend List Integer
euclq: Integer
if one?(snum) then
euclq:=1
else
  euclq := s.quotient
  -- sMult est la somme des multiplicit des points infiniment
  -- voisins par une trans. quadratique
  sMult: NNI := ( ( euclq - 1 ) * ee.i2 ) pretend NNI
  -- extMult est egal a la plus grande puissance de X que l'on peut
  --extraire de la transformee.
extMult := (ee.i1 + ee.i2 * euclq) pretend NonNegativeInteger
ch: BLMET
trStr:=BlUpRing
listBlRec: List blowUpRec
`zero?(s.reste ) =>
  ch:= createHN( i1 , i2 , chartCoord actChart, euclq , s.reste , _
    false , s.type)$BLMET
  trStr:= quadTransform(crb, extMult , ch )
  listBlRec:= [ [trStr,origin()$AFP,ch,actualExtension ]$blowUpRec ]
  [ sMult , listBlRec ]$recStr
polEdge := reduce( "+" , edge )
unipol:= univariate( replaceVarByOne( polEdge , i1 )$PackPoly )$PackPoly
recOfZeros:= distinguishedRootsOf( unipol , actualExtension )$RFP(K)
degExt:=recOfZeros.extDegree
`one?(degExt) =>
  print(("You need an extension of degree")::OutputForm)
  print(degExt::OutputForm)
  error("Have a nice day")
listOfZeroes:List K:= [ z for z in recOfZeros.zeros | ^zero?(z) ]
empty? listOfZeroes => _
  error " The curve is not absolutely irreducible since the Newton polygon has no sides "
ch:=_
  createHN( i1 , i2, chartCoord actChart, euclq, 0, false, s.type)$BLMET
lsTr:BlUpRing:= quadTransform(crb, extMult , ch )
lAff:List AFP:=[makeAff([ 0$K, z]:: List K , ch) for z in listOfZeroes ]
listBlRec := [ [ lsTr,p,ch,maxOf( actualExtension , z) ]$blowUpRec_
    for p in lAff for z in listOfZeroes ]
  [sMult, listBlRec ]$recStr

blowExp(exp,mult,chart)== -- CHH
  zero?( excepCoord chart) => exp
  lexp:List NNI:=parts(exp)
  ch1:Integer:= excepCoord chart
  ch2:Integer:= transCoord chart
  e1:Integer := lexp(ch1) pretend Integer
  e2:Integer := lexp(ch2) pretend Integer
  quotVal:Integer := quotValuation chart
  lbexp:=[0,0] :: List(NNI)
  lbexp(ch1):= ( e1 + quotVal * e2 - mult ) pretend NonNegativeInteger
  lbexp(ch2):=lexp(ch2)
  directProduct(vector(lbexp)$Vector(NNI))$E2

quadTransform(pol,mult,chart)== -- CHH
  mapExponents(blowExp(#1,mult,chart),pol)

polyRingToBlUpRing(pol,mult,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  ll:= [ d.i for i in 1..3 | ^( i = chartCoord(chart) ) ]

CHAPTER 3.  CHAPTER B
e:= directProduct( vector( ll)$Vector(NNI) )$E2
monomial(lc , e )$BlUpRing + polyRingToBlUpRing( reductum pol, chart )

bingToPolyRing(pol,chart) ==
zero? pol => 0
lc:= leadingCoefficient pol
d:=entries degree pol
nV:= chartCoord chart
ll:List NNI:=
nV = 1 => [ 0$NNI , d.1 , d.2 ]
nV = 2 => [ d.1 , 0$NNI , d.2 ]
[d.1 , d.2 , 0$NNI ]
e:= directProduct( vector( ll)$Vector(NNI) )$E
monomial(lc , e )$PolyRing + biringToPolyRing( reductum pol, chart )

applyTransform(pol,chart) ==
bingToPolyRing( quadTransform( polyRingToBlUpRing( pol, chart ),_,
0 , chart), chart )

-- K has PseudoAlgebraicClosureOfFiniteFieldCategory => maxTower([a,b])$K
-- K has PseudoAlgebraicClosureOfRationalNumberCategory => maxTower([a,b])$K
maxOf(a:K,b:K):K ==
K has PseudoAlgebraicClosureOfPerfectFieldCategory => maxTower([a,b])$K
1$K

———

— BLUPPACK.dotabb —-

"BLUPPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BLUPPACK"]
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"NPOLYGON" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPOLYGON"]
"BLUPPACK" -> "RFP"
"BLUPPACK" -> "NPOLYGON"

———

package BOUNDZRO BoundIntegerRoots

— BoundIntegerRoots.input —

)set break resume
)sys rm -f BoundIntegerRoots.output
)spool BoundIntegerRoots.output
)set message test on
3. CHAPTER B

)set message auto off
)clear all

--S 1 of 1
)show BoundIntegerRoots
--E 1

)spool
)lisp (bye)

---

BoundIntegerRoots .help —

BoundIntegerRoots examples

BoundIntegerRoots provides functions to find lower bounds on the integer roots of a polynomial.

See Also:
  o )show BoundIntegerRoots

---

BoundIntegerRoots (BOUNDZRO)

Exports:
  integerBound

---

package BOUNDZRO BoundIntegerRoots ---
)abbrev package BOUNDZRO BoundIntegerRoots
++ Author: Manuel Bronstein
++ Date Created: 11 March 1991
++ Date Last Updated: 18 November 1991
++ Description:
++ \spadtype{BoundIntegerRoots} provides functions to
++ find lower bounds on the integer roots of a polynomial.

BoundIntegerRoots(F, UP): Exports == Implementation where
  F : Join(Field, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  Z ==> Integer
  Q ==> Fraction Z
  K ==> Kernel F
  UPQ ==> SparseUnivariatePolynomial Q
  ALGOP ==> "%alg"

Exports ==> with
  integerBound: UP -> Z
  ++ integerBound(p) returns a lower bound on the negative integer
  ++ roots of p, and 0 if p has no negative integer roots.

Implementation ==> add
  import RationalFactorize(UPQ)
  import UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)

  qbound : (UP, UPQ) -> Z
  zroot1 : UP -> Z
  qzroot1: UPQ -> Z
  negint : Q -> Z

-- returns 0 if p has no integer root < 0, its negative integer root otherwise
qzroot1 p == negint(- leadingCoefficient(reductum p) / leadingCoefficient p)

-- returns 0 if p has no integer root < 0, its negative integer root otherwise
zroot1 p ==
  z := - leadingCoefficient(reductum p) / leadingCoefficient p
  (r := retractIfCan(z)@Union(Q, "failed"); case Q => negint(r::Q)) 0

-- returns 0 if r is not a negative integer, r otherwise
negint r ==
  (u := retractIfCan(r)@Union(Z, "failed"); case Z) and (u::Z < 0) => u::Z 0

if F has ExpressionSpace then
  bringDown: F -> Q

-- the random substitution used by bringDown is NOT always a ring-homorphism
-- (because of potential algebraic kernels), but is ALWAYS a Z-linear map.
-- this guarantees that bringing down the coefficients of \((x + n) q(x)\) for an
-- integer \(n\) yields a polynomial \(h(x)\) which is divisible by \(x + n\)
-- the only problem is that evaluating with random numbers can cause a
-- division by 0. We should really be able to trap this error later and
-- reevaluate with a new set of random numbers  MB 11/91

\[
\text{bringDown } f ==
\t t := \text{tower } f
\text{ retract eval}(f, t, [\text{random()}$\mathbb{Q}$ :: \(F\) for \(k\) in \(t\)])
\]

\[
\text{integerBound } p ==
\quad \text{one? degree } p \Rightarrow z\text{root1 } p
\quad (\text{degree } p) = 1 \Rightarrow z\text{root1 } p
\quad q1 := \text{map(bringDown, } p)
\quad q2 := \text{map(bringDown, } p)
\quad q\text{bound}(p, \gcd(q1, q2))
\]

\[
\text{else integerBound } p ==
\quad \text{one? degree } p \Rightarrow z\text{root1 } p
\quad (\text{degree } p) = 1 \Rightarrow z\text{root1 } p
\quad q\text{bound}(p, \text{map}((z1:F)\!::\!\mathbb{Q} \mapsto \text{retract}(z1)@\mathbb{Q}, p))
\]

-- we can probably do better here (i.e. without factoring)
\[
q\text{bound}(p, q) ==
\quad \text{bound:} \mathbb{Z} := 0
\quad \text{for rec in factors factor } q \text{ repeat}
\quad \quad \text{if one??(degree(rec.factor)) and ((r := qz\text{root1}(rec.factor)) < bound)
\quad \quad \quad \text{if ((degree(rec.factor)) = 1) and ((r := qz\text{root1}(rec.factor)) < bound)
\quad \quad \quad \quad \text{and zero? p(r::\mathbb{Q}::F) then bound := r)
\quad bound}
\]

———

— BOUNDZRO.dotabb ——

"BOUNDZRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BOUNDZRO"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BOUNDZRO" -> "PFECAT"

———

package BRILL BrillhartTests

— BrillhartTests.input ——
)set break resume
)sys rm -f BrillhartTests.output
)spool BrillhartTests.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BrillhartTests
--E 1

)spool
)lisp (bye)

---

— BrillhartTests.help —

====================================================================
BrillhartTests examples
====================================================================

This package has no description

[1] John Brillhart, Note on Irreducibility Testing,
[3] John Brillhart, On the Euler and Bernoulli polynomials,

See Also:
o )show BrillhartTests

---
BrillhartTests (BRILL)

Exports:

noLinearFactor? brillhartIrreducible? brillhartTrials

— package BRILL BrillhartTests —

)abbrev package BRILL BrillhartTests
++ Author: Frederic Lehobey, James H. Davenport
++ Date Created: 28 June 1994
++ Date Last Updated: 11 July 1997
++ References:
++ [1] John Brillhart, Note on Irreducibility Testing,
++ [3] John Brillhart, On the Euler and Bernoulli polynomials,
++ Description:
++ This package has no description

BrillhartTests(UP): Exports == Implementation where
N ==> NonNegativeInteger
Z ==> Integer
UP: UnivariatePolynomialCategory Z

Exports == with

brillhartIrreducible?: UP -> Boolean -- See [1]
  ++ brillhartIrreducible?(p) returns \spad{true} if p can be shown to be
  ++ irreducible by a remark of Brillhart, \spad{false} is inconclusive.
brillhartIrreducible?: (UP,Boolean) -> Boolean -- See [1]
  ++ brillhartIrreducible?(p,noLinears) returns \spad{true} if p can be
  ++ shown to be irreducible by a remark of Brillhart, \spad{false} else.
  ++ If noLinears is \spad{true}, we are being told p has no linear factors
  ++ \spad{false} does not mean that p is reducible.
brillhartTrials: () -> N
  ++ brillhartTrials() returns the number of tests in
  ++ \spadfun{brillhartIrreducible?}.
brillhartTrials: N -> N
  ++ brillhartTrials(n) sets to n the number of tests in
  ++ \spadfun{brillhartIrreducible?} and returns the previous value.
noLinearFactor?: UP -> Boolean -- See [3] p. 47
  ++ noLinearFactor?(p) returns \spad{true} if p can be shown to have no
  ++ linear factor by a theorem of Lehmer, \spad{false} else. I insist on
  ++ the fact that \spad{false} does not mean that p has a linear factor.

Implementation ==> add

import GaloisGroupFactorizationUtilities(Z,UP,Float)

squaredPolynomial(p:UP):Boolean ==
  d := degree p
do = 0 => true
  odd? d => false
  squaredPolynomial reductum p

primeEnough?(n:Z,b:Z):Boolean ==
  -- checks if n is prime, with the possible exception of
  -- factors whose product is at most b
  import Float
  bb: Float := b::Float
  for i in 2..b repeat
    while (d:= n exquo i) case Integer repeat
      n:=d::Integer
      bb:=bb / i::Float
    bb < 1$Float => return false
    --- we over-divided, so it can’t be prime
    prime? n

brillharttrials: N := 6
brillhartTrials():N == brillharttrials

brillhartTrials(n:N):N ==
  (brillharttrials,n) := (n,brillharttrials)
n
brillhartIrreducible?(p:UP):Boolean ==
  brillhartIrreducible?(p,noLinearFactor? p)

  zero? brillharttrials => false
  origBound := (largeEnough := rootBound(p)+1)
  -- see remarks 2 and 4
  even0 := even? coefficient(p,0)
even1 := even? p(1)
polyx2 := squaredPolynomial(p)
prime? p(largeEnough) => true
  not polyx2 and prime? p(-largeEnough) => true
-- one? brillharttrials => false
(brillharttrials = 1) => false
largeEnough := largeEnough+1
primeEnough?(p(largeEnough),if noLinears then 4 else 2) => true
not polyx2 and
primeEnough?(p(-largeEnough),if noLinears then 4 else 2) => true
if odd? largeEnough then
  if even0 then largeEnough := largeEnough+1
else
  if even1 then largeEnough := largeEnough+1
count :=(if polyx2 then 2 else 1)*(brillharttrials-2)+largeEnough
for i in (largeEnough+1)..count repeat
  small := if noLinears then (i-origBound)**2 else (i-origBound)
  primeEnough?(p(i),small) => return true
  not polyx2 and primeEnough?(p(-i),small) => return true
false

noLinearFactor?(p:UP):Boolean ==
  (odd? leadingCoefficient p) and (odd? coefficient(p,0)) and (odd? p(1))

———

— BRILL.dotabb ——

"BRILL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BRILL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BRILL" -> "PFECAT"

———
Chapter 4

Chapter C

package CARTEN2 CartesianTensorFunctions2

— CartesianTensorFunctions2.input —

)set break resume
)sys rm -f CartesianTensorFunctions2.output
)spool CartesianTensorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CartesianTensorFunctions2
--E 1

)spool
)lisp (bye)

— CartesianTensorFunctions2.help —

====================================================================
CartesianTensorFunctions2 examples
====================================================================

This package provides functions to enable conversion of tensors
given conversion of the components.

See Also:
CHAPTER 4. CHAPTER C

o )show CartesianTensorFunctions2

---

CartesianTensorFunctions2 (CARTEN2)

Exports:
map  reshape

— package CARTEN2 CartesianTensorFunctions2 —

)abbrev package CARTEN2 CartesianTensorFunctions2
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides functions to enable conversion of tensors
++ given conversion of the components.

CartesianTensorFunctions2(minix, dim, S, T): CTPcat == CTPdef where

minix: Integer
dim:  NonNegativeInteger
S,  T:  CommutativeRing
CS  ==>  CartesianTensor(minix, dim, S)
CT  ==>  CartesianTensor(minix, dim, T)

CTPcat == with

reshape: (List T, CS) -> CT
++ reshape(lt,ts) organizes the list of components lt into
++ a tensor with the same shape as ts.
map: (S->T,  CS) -> CT
++ map(f,ts) does a componentwise conversion of the tensor ts
++ to a tensor with components of type T.
package CHVAR ChangeOfVariable

--- ChangeOfVariable.input ---

)set break resume
)sys rm -f ChangeOfVariable.output
)spool ChangeOfVariable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ChangeOfVariable
--E 1

)spool
)lisp (bye)

--- ChangeOfVariable.help ---

====================================================================
ChangeOfVariable examples
====================================================================

Tools to send a point to infinity on an algebraic curve.

See Also:
ChangeOfVariable (CHVAR)

Exports:
chvar    eval    goodPoint    mkIntegral
radPoly  rootPoly  

— package CHVAR ChangeOfVariable —

)abbrev package CHVAR ChangeOfVariable
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 Feb 1990
++ Description:
++ Tools to send a point to infinity on an algebraic curve.

ChangeOfVariable(F, UP, UPUP): Exports == Implementation where
  F : UniqueFactorizationDomain
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP

  N ==> NonNegativeInteger
  Z ==> Integer
  Q ==> Fraction Z
  RF ==> Fraction UP

Exports ==> with
  mkIntegral: UPUP -> Record(coef:RF, poly:UPUP)
    ++ mkIntegral(p(x,y)) returns \spad{[c(x), q(x,z)]} such that
    ++ \spad{z = c * y} is integral.
    ++ The algebraic relation between x and y is \spad{p(x, y) = 0}. 
++ The algebraic relation between $x$ and $z$ is \spad{q(x, z) = 0}.

\begin{verbatim}
radPoly : UPUP -> Union(Record(radicand:RF, deg:N), "failed")
++ radPoly(p(x, y)) returns \spad{[c(x), n]} if $p$ is of the form
++ \spad{y**n - c(x)}, "failed" otherwise.

rootPoly : (RF, N) -> Record(exponent: N, coef:RF, radicand:UP)
++ rootPoly(g, n) returns \spad{[m, c, P]} such that
++ \spad{c * g ** (1/n) = P ** (1/m)}
++ thus if \spad{y**n = g}, then \spad{x**m = P}
++ where \spad{z = c * y}.

goodPoint : (UPUP,UPUP) -> F
++ goodPoint(p, q) returns an integer $a$ such that $a$ is neither
++ a pole of \spad{p(x,y)} nor a branch point of \spad{q(x,y) = 0}.

eval : (UPUP, RF, RF) -> UPUP
++ eval(p(x,y), f(x), g(x)) returns \spad{p(f(x), y * g(x))}.

chvar : (UPUP,UPUP) -> Record(func:UPUP,poly:UPUP,c1:RF,c2:RF,deg:N)
++ chvar(f(x,y), p(x,y)) returns
++ \spad{[g(z,t), q(z,t), c1(z), c2(z), n]} such that under the change of variable
++ \spad{x = c1(z)}, \spad{y = t * c2(z)},
++ one gets \spad{f(x,y) = g(z,t)}.
++ The algebraic relation between $x$ and $y$ is \spad{p(x, y) = 0}.
++ The algebraic relation between $z$ and $t$ is \spad{q(z, t) = 0}.

Implementation ==> add

import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

algPoly : UPUP -> Record(coef:RF, poly:UPUP)
RPrim : (UP, UP, UPUP) -> Record(coef:RF, poly:UPUP)
good? : (F, UP, UP) -> Boolean
infIntegral?: (UPUP, UPUP) -> Boolean

eval(p, x, y) == map(s +-> s(x), p) monomial(y, 1)
good?(a, p, q) == p(a) ^= 0 and q(a) ^= 0

algPoly p ==
  ground?(a:= retract(leadingCoefficient(q:=clearDenominator p))@UP)
  => RPrim(1, a, q)
  c := d := squareFreePart a
  q := clearDenominator q monomial(inv(d::RF), 1)
  while not ground?(a := retract(leadingCoefficient q)@UP) repeat
    c := c * (d := gcd(a, d))
    q := clearDenominator q monomial(inv(d::RF), 1)
  RPrim(c, a, q)

RPrim(c, a, q) ==
  -- one? a => [c::RF, q]
  -- a = 1 => [c::RF, q]
  -- always makes the algebraic integral, but does not send a point to infinity

--
-- if the integrand does not have a pole there (in the case of an nth-root)
chvar(f, modulus) ==
  r1 := mkIntegral modulus
  f1 := f monomial(r1inv := inv(r1.coef), 1)
infIntegral?(f1, r1.poly) =>
  [f1, r1.poly, monomial(1,1)$RF, r1inv, degree(retract(r1.coef)@RF)]
  x := (a:= goodPoint(f1,r1.poly))::RF + inv(monomial(1,1)::RF)
  r2c:= retract((r2 := mkIntegral map(s+->s(x), r1.poly)).coef)@RF
  t := inv((monomial(1, 1)$RF - a::RF))
  [- inv(monomial(1, 2)$RF) * eval(f1, x, inv(r2.coef)),
    r2.poly, t, r1.coef * r2c t, degree r2c]

-- returns true if y is an n-th root, and it can be guaranteed that p(x,y)dx
-- is integral at infinity
-- expects y to be integral.
infiniteIntegral?(p, modulus) ==
  (r := radPoly modulus) case "failed" => false
  ninv := inv(r.deg::Q)
  degy:Q := degree(retract(r.radicand)@RF) * ninv
  degp:Q := 0
  while p ^= 0 repeat
    c := leadingCoefficient p
    degp := max(degp,
                (2 + degree(numer c)::Z - degree(denom c)::Z)::Q + degree(p) * degy)
    p := reductum p
  degp <= ninv

mkIntegral p ==
  (r := radPoly p) case "failed" => algPoly p
  rp := rootPoly(r.radicand, r.deg)
  [rp.coef, monomial(1, rp.exponent)$RF - rp.radicand::RF]

goodPoint(p, modulus) ==
  q :=
    (r := radPoly modulus) case "failed" =>
      retract(radicand)@UP
    retract(resultant(modulus, differentiate modulus))@UP
  d := commonDenominator p
  for i in 0.. repeat
    good?(a := i::F, q, d) => return a
    good?(-a, q, d) => return -a

radPoly p ==
  (r := retractIfCan(reductum p)@Union(RF, "failed")) case "failed"
  => "failed"
  [- (r::RF), degree p]

-- we have y**m = g(x) = n(x)/d(x), so if we can write
-- (n(x) * d(x)**(m-1)) * (1/m) = c(x) * P(x) ** (1/n)
-- then z**q = P(x) where z = (d(x) / c(x)) * y
`rootPoly(g, m) ==
zero? g => error "Should not happen"
pr := nthRoot(squareFree((numer g) * (d := denom g) ** (m-1)::N),
       m)$FactoredFunctions(UP)
[pr.exponent, d / pr.coef, */(pr.radicand)]

---

package CPIMA CharacteristicPolynomialInMonogenicalAlgebra

--- CharacteristicPolynomialInMonogenicalAlgebra.input ---

)`set break resume
)`sys rm -f CharacteristicPolynomialInMonogenicalAlgebra.output
)`spool CharacteristicPolynomialInMonogenicalAlgebra.output
)`set message test on
)`set message auto off
)`clear all

--S 1 of 1
)`show CharacteristicPolynomialInMonogenicalAlgebra

--E 1

)`spool
)`lisp (bye)

---

--- CharacteristicPolynomialInMonogenicalAlgebra.help ---

====================================================================
CharacteristicPolynomialInMonogenicalAlgebra examples
====================================================================
This package implements characteristicPolynomials for monogenic algebras using resultants

See Also:
- )show CharacteristicPolynomialInMonogenousAlgebra
import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, Pol(PolR))
XtoY(Q : PolR) : Pol(PolR) == map(x+->monomial(x, 0), Q)

P : Pol(PolR) := XtoY(definingPolynomial()$E)
X : Pol(PolR) := monomial(monomial(1, 1)$PolR, 0)

characteristicPolynomial(x : E) : PolR ==
  Qx : PolR := lift(x)
  -- on utilise le fait que resultant_Y (P(Y), X - Qx(Y))
  return resultant(P, X - XtoY(Qx))

---

package CHARPOL CharacteristicPolynomialPackage

--- CharacteristicPolynomialPackage.input ---

)set break resume
)sys rm -f CharacteristicPolynomialPackage.output
)spool CharacteristicPolynomialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CharacteristicPolynomialPackage
--E 1

)spool
)lisp (bye)

---

--- CharacteristicPolynomialPackage.help ---

====================================================================
CharacteristicPolynomialPackage examples
===============================================

This package provides a characteristicPolynomial function for any matrix over a commutative ring.

See Also:

- )show CharacteristicPolynomialPackage

---

CharacteristicPolynomialPackage (CHARPOL)

Exports:
characteristicPolynomial

— package CHARPOL CharacteristicPolynomialPackage —

)abbrev package CHARPOL CharacteristicPolynomialPackage
++ Author: Barry Trager
++ Description:
++ This package provides a characteristicPolynomial function
++ for any matrix over a commutative ring.

CharacteristicPolynomialPackage(R:CommutativeRing): C == T where
PI ==> PositiveInteger
M ==> Matrix R
C == with
characteristicPolynomial: (M, R) -> R
    ++ characteristicPolynomial(m,r) computes the characteristic
    ++ polynomial of the matrix m evaluated at the point r.
    ++ In particular, if r is the polynomial 'x, then it returns
    ++ the characteristic polynomial expressed as a polynomial in 'x.
T == add

---- characteristic polynomial ----
characteristicPolynomial(A:M,v:R) : R ==
dimA :PI := (nrows A):PI
if dimA ^= ncols A => error "The matrix is not square"
B:M:=zero(dimA,dimA)
for i in 1..dimA repeat
  for j in 1..dimA repeat B(i,j):=A(i,j)
  B(i,i) := B(i,i) - v
  determinant B

———

--- CHARPOL.dotabb ---
"CHARPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CHARPOL"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"CHARPOL" -> "BMODULE"

———

package IBACHIN ChineseRemainderToolsForIntegralBases

--- ChineseRemainderToolsForIntegralBases.input ---

)set break resume
)sys rm -f ChineseRemainderToolsForIntegralBases.output
)spool ChineseRemainderToolsForIntegralBases.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ChineseRemainderToolsForIntegralBases
--E 1

)spool
)lisp (bye)

———

--- ChineseRemainderToolsForIntegralBases.help ---
CHAPTER 4. CHAPTER C

ChineseRemainderToolsForIntegralBases examples

This package has no description

See Also:
  o )show ChineseRemainderToolsForIntegralBases

ChineseRemainderToolsForIntegralBases (IBACHIN)

Exports:
  chineseRemainder  factorList  listConjugateBases

package IBACHIN ChineseRemainderToolsForIntegralBases —

)abbrev package IBACHIN ChineseRemainderToolsForIntegralBases
++ Author: Clifton Williamson
++ Date Created: 9 August 1993
++ Date Last Updated: 3 December 1993
++ Description:
++ This package has no description

ChineseRemainderToolsForIntegralBases(K,R,UP): Exports == Implementation where
  K : FiniteFieldCategory
  R : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R

  DDFACT => DistinctDegreeFactorize
  I => Integer
  L => List
 PACKAGE IBACHIN CHINESEREMAINDERTOOLSFORINTEGRALBASES

L2 ==> ListFunctions2
Mat ==> Matrix R
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
Q ==> Fraction R
SAE ==> SimpleAlgebraicExtension
SUP ==> SparseUnivariatePolynomial
SUP2 ==> SparseUnivariatePolynomialFunctions2
Result ==> Record(basis: Mat, basisDen: R, basisInv: Mat)

Exports ==> with
factorList: (K,NNI,NNI,NNI) -> L SUP K
  + factorList(k,n,m,j) undocumented

listConjugateBases: (Result,NNI,NNI) -> List Result
  + listConjugateBases(bas,q,n) returns the list
    \spad{[bas,bas^{Frob},bas^{(Frob^2)},...bas^{(Frob^{(n-1)})}]}, where
    \spad{Frob} raises the coefficients of all polynomials
    appearing in the basis \spad{bas} to the \spad{q}th power.

chineseRemainder: (List UP, List Result, NNI) -> Result
  + chineseRemainder(lu,lr,n) undocumented

Implementation ==> add
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

applyFrobToMatrix: (Matrix R,NNI) -> Matrix R
applyFrobToMatrix(mat,q) ==
  -- raises the coefficients of the polynomial entries of 'mat'
  -- to the qth power
  m := nrows mat; n := ncols mat
  ans : Matrix R := new(m,n,0)
  for i in 1..m repeat for j in 1..n repeat
    qsetelt_!(ans,i,j,map((k1:K):K +-> k1 ** q,qelt(mat,i,j)))
  ans

listConjugateBases(bas,q,n) ==
  outList : List Result := list bas
  b := bas.basis; bInv := bas.basisInv; bDen := bas.basisDen
  for i in 1..(n-1) repeat
    b := applyFrobToMatrix(b,q)
bInv := applyFrobToMatrix(bInv,q)
bDen := map((k1:K):K +-> k1 ** q,bDen)
  newBasis : Result := [b,bDen,bInv]
  outList := concat(newBasis,outList)
  reverse_! outList

factorList(a,q,n,k) ==
  coef : SUP K := monomial(a,0); xx : SUP K := monomial(1,1)
outList : L SUP K := list((xx - coef)**k)
for i in 1..(n-1) repeat
  coef := coef ** q
  outList := concat((xx - coef)**k, outList)
reverse_! outList

basisInfoToPolys: (Mat,R,R) -> L UP
basisInfoToPolys(mat,lcm,den) ==
n := nrows(mat) :: I; n1 := n - 1
outList : L UP := empty()
for i in 1..n repeat
  pp : UP := 0
  for j in 0..n1 repeat
    pp := pp + monomial((lcm quo den) * qelt(mat,i,j+1),j)
  outList := concat(pp, outList)
reverse_! outList

basesToPolyLists: (L Result,R) -> L L UP
basesToPolyLists(basisList,lcm) ==
  [basisInfoToPolys(b.basis,lcm,b.basisDen) for b in basisList]

OUT ==> OutputForm

approximateExtendedEuclidean: (UP,UP,R,NNI) -> Record(coef1:UP,coef2:UP)
aproximateExtendedEuclidean(f,g,p,n) ==
-- f and g are monic and relatively prime (mod p)
-- function returns [coef1,coef2] such that
-- coef1 * f + coef2 * g = 1 (mod p^n)
sae := SAE(K,R,p)
fSUP : SUP R := makeSUP f; gSUP : SUP R := makeSUP g
fBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,fSUP)$SUP2(R,sae)
gBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,gSUP)$SUP2(R,sae)
ee := extendedEuclidean(fBar,gBar)
-- not one?(ee.generator) =>
not (ee.generator = 1) =>
  error "polynomials aren't relatively prime"
ss1 := ee.coef1; tt1 := ee.coef2
s1 : SUP R := map((z1:sae):R +-> convert(z1)@SUP2(sae,R)); s := s1
t1 : SUP R := map((z1:sae):R +-> convert(z1)@SUP2(sae,R),tt1)$SUP2(R,sae);
t := t1
pPower := p
for i in 2..n repeat
  num := 1 - s * fSUP - t * gSUP
  rhs := (num exquo pPower) :: SUP R
  sigma := map((r1:R):R +-> r1 rem p,s1*rhs);
  tau := map((r1:R):R +-> r1 rem p,t1*rhs)
  s := s + pPower * sigma; t := t + pPower * tau
  quorem := monicDivide(s,gSUP)
pPower := pPower * p
  s := map((r1:R):R +-> r1 rem pPower,quorem.remainder)
  t := map((r1:R):R +-> r1 rem pPower,t + fSUP * (quorem.quotient))
[unmakeSUP s, unmakeSUP t]

-- mapChineseToList: (L SUP Q, L SUP Q, I) -> L SUP Q
-- mapChineseToList(list, polyList, i) ==
mapChineseToList: (L UP, L UP, I, R) -> L UP
mapChineseToList(list, polyList, i, den) ==
-- 'polyList' consists of MONIC polynomials
-- computes a polynomial p such that p = pp (modulo polyList[i])
-- and p = 0 (modulo polyList[j]) for j /= i for each 'pp' in 'list'
-- create polynomials
q : UP := 1
for j in 1..(i-1) repeat
  q := q * first polyList
  polyList := rest polyList
p := first polyList
polyList := rest polyList
for j in (i+1).. while not empty? polyList repeat
  q := q * first polyList
  polyList := rest polyList
-- p := map((numer(#1) rem den)/1, p)
-- q := map((numer(#1) rem den)/1, q)
-- 'den' is a power of an irreducible polynomial
--!! make this computation more efficient!!
factoredDen := factor(den)$DDFACT(K,R)
prime := nthFactor(factoredDen, 1)
n := nthExponent(factoredDen, 1) :: NNI
invPoly := approximateExtendedEuclidean(q, p, prime, n).coef1
-- monicDivide may be inefficient?
[monicDivide(pp * invPoly, q, p, q).remainder for pp in list]

polyListToMatrix: (L UP, NNI) -> Mat
polyListToMatrix(polyList, n) ==
mat : Mat := new(n, n, 0)
for i in 1..n for poly in polyList repeat
  while not zero? poly repeat
    mat(i, degree(poly) + 1) := leadingCoefficient poly
  poly := reductum poly
mat

chineseRemainder(factors, factorBases, n) ==
denLCM : R := reduce("lcm", [base.basisDen for base in factorBases])
denLCM = 1 => [scalarMatrix(n, 1, 1), scalarMatrix(n, 1)]
-- compute local basis polynomials with denominators cleared
factorBasisPolyLists := basesToPolyLists(factorBases, denLCM)
-- use Chinese remainder to compute basis polynomials w/o denominators
basisPolyLists : L L UP := empty()
for i in 1.. for pList in factorBasisPolyLists repeat
  polyList := mapChineseToList(pList, factors, i, denLCM)
  basisPolyLists := concat(polyList, basisPolyLists)
basisPolys := concat reverse_! basisPolyLists
\[ \text{mat} := \text{squareTop rowEchelon(polyListToMatrix(basisPolys,n),denLCM)} \]
\[ \text{matInv} := \text{UpTriBddDenomInv(mat,denLCM)} \]
\[ [\text{mat,denLCM,matInv}] \]

---

**IBACHIN.dotabb**

"IBACHIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBACHIN"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"IBACHIN" -> "MONOGEN"

---

**package CVMP CoerceVectorMatrixPackage**

**CoerceVectorMatrixPackage.input**

```lisp
)set break resume
)sys rm -f CoerceVectorMatrixPackage.output
)spool CoerceVectorMatrixPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CoerceVectorMatrixPackage
--E 1

)spool
)lisp (bye)
```

---

**CoerceVectorMatrixPackage.help**

====================================================================
CoerceVectorMatrixPackage examples
====================================================================

CoerceVectorMatrixPackage is an unexposed, technical package
for data conversions
CoerceVectorMatrixPackage (CVMP)

Exports:
coerce  coerceP

--- package CVMP CoerceVectorMatrixPackage ---

)abbrev package CVMP CoerceVectorMatrixPackage
++ Author: J. Grabmeier
++ Date Created: 26 June 1991
++ Date Last Updated: 26 June 1991
++ Description:
++ CoerceVectorMatrixPackage is an unexposed, technical package
++ for data conversions

CoerceVectorMatrixPackage(R : CommutativeRing): public == private where
M2FP ==> MatrixCategoryFunctions2(R, Vector R, Vector R, Matrix R, _
        Fraction Polynomial R, Vector Fraction Polynomial R, _
        Vector Fraction Polynomial R, Matrix Fraction Polynomial R)
public ==
coerceP: Vector Matrix R -> Vector Matrix Polynomial R
  ++ coerceP(v) coerces a vector v with entries in \spadtype{Matrix R}
  ++ as vector over \spadtype{Matrix Polynomial R}
coerce: Vector Matrix R -> Vector Matrix Fraction Polynomial R
  ++ coerce(v) coerces a vector v with entries in \spadtype{Matrix R}
  ++ as vector over \spadtype{Matrix Fraction Polynomial R}
private ==> add

imbedFP : R -> Fraction Polynomial R
imbedFP r == (r:: Polynomial R) :: Fraction Polynomial R

imbedP : R -> Polynomial R
imbedP r == (r:: Polynomial R)

coerceP(g:Vector Matrix R) : Vector Matrix Polynomial R ==
m2 : Matrix Polynomial R
lim : List Matrix R := entries g
l: List Matrix Polynomial R := []
for m in lim repeat
  m2 := map(imbedP,m)$M2P
  l := cons(m2,1)
vector reverse l

coerce(g:Vector Matrix R) : Vector Matrix Fraction Polynomial R ==
m3 : Matrix Fraction Polynomial R
lim : List Matrix R := entries g
l: List Matrix Fraction Polynomial R := []
for m in lim repeat
  m3 := map(imbedFP,m)$M2FP
  l := cons(m3,1)
vector reverse l

package COMBF CombinatorialFunction

---

"CVMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CVMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CVMP" -> "PFECAT"

---

)set break resume
)sys rm -f CombinatorialFunction.output
)spool CombinatorialFunction.output
)set message test on
```lisp
)set message auto off
)clear all

-- S 1 of 6
f := operator 'f
-- R
-- R
-- R (1) f
-- R

Type: BasicOperator
-- E 1

-- S 2 of 6
D(product(f(i,x),i=1..m),x)
-- R
-- R
-- R m m f (i,x)
-- R ++++ --- ,2
-- R (2) | | f(i,x)> -------
-- R | | +++ f(i,x)
-- R i= 1 i= 1

Type: Expression(Integer)
-- E 2

-- S 3 of 6
)set expose add constructor OutputForm
-- R
-- R OutputForm is already explicitly exposed in frame frame0
-- E 3

-- S 4 of 6
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
-- R
-- R

Type: Void
-- E 4

-- S 5 of 6
displayRow(n)==output center blankSeparate pascalRow(n)
-- R
-- R

Type: Void
-- E 5

-- S 6 of 6
for i in 0..7 repeat displayRow i
-- R
-- R Compiling function pascalRow with type NonNegativeInteger -> List(
-- R OutputForm)
-- R Compiling function displayRow with type NonNegativeInteger -> Void
-- R 1
-- R 1 1
-- R 1 2 1
-- R 1 3 3 1
```
CHAPTER C

---R

---R

---R

---R

---R

---R

---E 6

)spool
)lisp (bye)

---

— CombinatorialFunction.help —

====================================================================
CombinatorialFunction examples
====================================================================

Provides combinatorial functions over an integral domain.

\( f := \text{operator 'f} \)

(1) \( f \)

\( \text{Type: BasicOperator} \)

\( D(\text{product}(f(i,x),i=1..m),x) \)

\( \begin{array}{c}
\mbox{m} \\
\mbox{m} \\
\end{array} \)
\( \begin{array}{c}
f(i,x) \\
\mbox{---+} \\
\mbox{++-+} \\
\end{array} \)
\( \mbox{,2} \)

(2) \( \begin{array}{c}
\mbox{f(i,x)>} \\
\mbox{---+} \\
i=1 \\
i=1 \\
\end{array} \)

\( \text{Type: Expression Integer} \)

The binomial\( (n, r) \) returns the number of subsets of \( r \) objects taken among \( n \) objects, i.e. \( n!/(r! \cdot (n-r)! \)

The binomial coefficients are the coefficients of the series expansion of a power of a binomial, that is

\( \begin{array}{c}
n \\
---+ \\
\mbox{n} \\
\mbox{/ n \ \ k} \\

> \mbox{x} = (1 + x) \\
---+ \\
\mbox{\ k /} \\
\mbox{k= 0} \\
\end{array} \)
This leads to the famous pascal triangle. First we expose the OutputForm domain, which is normally hidden, so we can use it to format the lines.

)set expose add constructor OutputForm

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:

\[
pascalRow(n) == \text{[right(binomial(n,i),4) for i in 0..n]}
\]

and now we format the whole line so that it looks centered:

\[
displayRow(n) == \text{output center blankSeparate pascalRow(n)}
\]

and we compute the triangle

\[
\text{for i in 0..7 repeat displayRow i}
\]

giving the pretty result:

```
Compiling function pascalRow with type NonNegativeInteger -> List OutputForm
Compiling function displayRow with type NonNegativeInteger -> Void

1
  1
  1 2 1
  1 3 3 1
  1 4 6 4 1
  1 5 10 10 5 1
  1 6 15 20 15 6 1
  1 7 21 35 35 21 7 1
```

See Also:
- )show CombinatorialFunction
- )d op binomial
- )show OutputForm
- )help set
CombinatorialFunction (COMBF)

Exports:
- belong?
- binomial
- factorial
- factorials
- iibinom
- iidprod
- iidsum
- iifact
- iiperm
- iipow
- ipow
- permutation
- product
- summation
- operator
- product
- summation
- ?**?

binomial

We currently simplify binomial coefficients only for non-negative integral second argument, using the formula
\[
\binom{n}{k} = \frac{1}{k!} \prod_{i=0}^{k-1} (n - i),
\]
except if the second argument is symbolic: in this case \(\binom{n}{n}\) is simplified to one.

Note that there are at least two different ways to define binomial coefficients for negative integral second argument. One way, particular suitable for combinatorics, is to set the binomial coefficient equal to zero for negative second argument. This is, partially, also the approach taken in combinat.spad, where we find

\[
\text{binomial}(n, m) ==
\begin{align*}
& n < 0 \text{ or } m < 0 \text{ or } m > n \Rightarrow 0 \\
& m = 0 \Rightarrow 1
\end{align*}
\]

Of course, here \(n\) and \(m\) are integers. This definition agrees with the recurrence
\[
\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}.
\]

Alternatively, one can use the formula
\[
\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}.
\]
and leave the case where \(k \in \mathbb{Z}, n \in \mathbb{Z}\) and \(k \leq n < 0\) undefined, since the limit does not exist in this case:

Since we then have that \(n - k + 1 \geq 1\), \(\Gamma(n - k + 1)\) is finite. So it is sufficient to consider \(\frac{\Gamma(n + 1)}{\Gamma(k + 1)}\). On the one hand, we have

\[
\lim_{n_0 \to n} \lim_{k_0 \to k} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)} = 0,
\]

since for any non-integral \(n_0\), \(\Gamma(n_0 + 1)\) is finite. On the other hand,

\[
\lim_{k_0 \to k} \lim_{n_0 \to n} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)}
\]

does not exist, since for non-integral \(k_0\), \(\Gamma(k_0 + 1)\) is finite while \(\Gamma(n_0 + 1)\) is unbounded.

However, since for \(k \in \mathbb{Z}, n \in \mathbb{Z}\) and \(0 < k < n\) both definitions agree, one could also combine them. This is what, for example, Mathematica does. It seems that MuPAD sets \(\text{binomial}(n,n)=1\) for all arguments \(n\), and returns \(\text{binomial}(-2, n)\) unevaluated. Provisos may help here.

dvsum and dvdsum

The dvsum and dvdsum operations implement differentiation of sums with and without bounds. Note that the function

\[
n \mapsto \sum_{k=1}^{n} f(k,n)
\]

is well defined only for integral values of \(n\) greater than or equal to zero. There is not even consensus how to define this function for \(n < 0\). Thus, it is not differentiable. Therefore, we need to check whether we erroneously are differentiating with respect to the upper bound or the lower bound, where the same reasoning holds.

Differentiating a sum with respect to its indexing variable correctly gives zero. This is due to the introduction of dummy variables in the internal representation of a sum: the operator \%defsum\ takes 5 arguments, namely

1. the summands, where each occurrence of the indexing variable is replaced by
2. the dummy variable,
3. the indexing variable,
4. the lower bound, and
5. the upper bound.
The dvprod and dvdprod operations implement differentiation of products with and without bounds. Note again, that we cannot even properly define products with bounds that are not integral.

To differentiate the product, we use Leibniz rule:

\[
\frac{d}{dx} \prod_{i=a}^{b} f(i, x) = \sum_{i=a}^{b} \frac{d}{dx} f(i, x) \prod_{i=a}^{b} f(i, x)
\]

There is one situation where this definition might produce wrong results, namely when the product is zero, but axiom failed to recognize it: in this case,

\[
\frac{d}{dx} f(i, x) / f(i, x)
\]

is undefined for some \(i\). However, I was not able to come up with an example. The alternative definition

\[
\frac{d}{dx} \prod_{i=a}^{b} f(i, x) = \sum_{i=a}^{b} \left( \frac{d}{dx} f(i, x) \right) \prod_{j=a, j \neq i}^{b} f(j, x)
\]

has the slight (display) problem that we would have to come up with a new index variable, which looks very ugly. Furthermore, it seems to me that more simplifications will occur with the first definition.

\[f := \text{operator 'f}
\]
\[D(\text{product}(f(i, x), i=1..m), x)\]

The dvpow2 operation implements the differentiation of the power operator \(\%\text{power}\) with respect to its second argument, i.e., the exponent. It uses the formula

\[
\frac{d}{dx} g(y)^x = \frac{d}{dx} e^{x \log g(y)} = \log g(y) g(y)^x.
\]

If \(g(y)\) equals zero, this formula is not valid, since the logarithm is not defined there. Although strictly speaking \(0^x\) is not differentiable at zero, we return zero for convenience.
++ Provides combinatorial functions over an integral domain.

CombinatorialFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R

OP ==> BasicOperator
K ==> Kernel F
SE ==> Symbol
O ==> OutputForm
SMP ==> SparseMultivariatePolynomial(R, K)
Z ==> Integer

POWER ==> "%power"::Symbol
OPEXP ==> "%exp"::Symbol
SPECIALDIFF ==> "%specialDiff"
SPECIALDISP ==> "%specialDisp"
SPECIALEQUAL ==> "%specialEqual"

Exports ==> with
  belong? : OP -> Boolean
    ++ belong?(op) is true if op is a combinatorial operator;
  operator : OP -> OP
    ++ operator(op) returns a copy of op with the domain-dependent
    ++ properties appropriate for F;
    ++ error if op is not a combinatorial operator;
  "**" : (F, F) -> F
    ++ a ** b is the formal exponential a**b;
  binomial : (F, F) -> F
    ++ binomial(n, r) returns the number of subsets of r objects
    ++ taken among n objects, i.e. n!/(r! * (n-r)!);
    ++
    ++X [binomial(5,i) for i in 0..5]
  permutation: (F, F) -> F
    ++ permutation(n, r) returns the number of permutations of
    ++ n objects taken r at a time, i.e. n!/(n-r)!
  factorial : F -> F
    ++ factorial(n) returns the factorial of n, i.e. n!;
  factorials : F -> F
    ++ factorials(f) rewrites the permutations and binomials in f
    ++ in terms of factorials;
  factorials : (F, SE) -> F
    ++ factorials(f, x) rewrites the permutations and binomials in f
    ++ involving x in terms of factorials;
  summation : (F, SE) -> F
    ++ summation(f(n), n) returns the formal sum S(n) which verifies
    ++ S(n+1) - S(n) = f(n);
  summation : (F, SegmentBinding F) -> F
    ++ summation(f(n), n = a..b) returns f(a) + ... + f(b) as a
    ++ formal sum;
product : (F, SE) -> F
++ product(f(n), n) returns the formal product P(n) which verifies
++ P(n+1)/P(n) = f(n);
product : (F, SegmentBinding F) -> F
++ product(f(n), n = a..b) returns f(a) * ... * f(b) as a
++ formal product;
ifact : F -> F
++ ifact(x) should be local but conditional;
iibinom : List F -> F
++ iibinom(l) should be local but conditional;
iiperm : List F -> F
++ iiperm(l) should be local but conditional;
iipow : List F -> F
++ iipow(l) should be local but conditional;
iidsum : List F -> F
++ iidsum(l) should be local but conditional;
iidprod : List F -> F
++ iidprod(l) should be local but conditional;

Implementation ==> add
ifact : F -> F
iiipow : List F -> F
iperm : List F -> F
iibinom : List F -> F
isum : List F -> F
idsum : List F -> F
iprod : List F -> F
idprod : List F -> F
dsum : List F -> O
ddsum : List F -> O
dprod : List F -> O
ddprod : List F -> O
equalsumprod : (K, K) -> Boolean
equaldsumprod : (K, K) -> Boolean
fourth : List F -> F
dvpow1 : List F -> F
dvpow2 : List F -> F
summand : List F -> F
dvsum : (List F, SE) -> F
dvdsum : (List F, SE) -> F
dvprod : (List F, SE) -> F
dvdprod : (List F, SE) -> F
facts : (F, List SE) -> F
K2fact : (K, List SE) -> F
smpfact : (SMP, List SE) -> F

-- This macro will be used in product and summation, both the 5 and 3
-- argument forms. It is used to introduce a dummy variable in place of the
-- summation index within the summands. This in turn is necessary to keep the
-- indexing variable local, circumventing problems, for example, with
-- differentiation.

dummy == new()$SE :: F

opfact := operator("factorial"::Symbol)$CommonOperators
opperm := operator("permutation"::Symbol)$CommonOperators
opbinom := operator("binomial"::Symbol)$CommonOperators
opsum := operator("summation"::Symbol)$CommonOperators
opdsum := operator("%defsum"::Symbol)$CommonOperators
opprod := operator("product"::Symbol)$CommonOperators
opdprod := operator("%defprod"::Symbol)$CommonOperators
oppow := operator(POWER::Symbol)$CommonOperators

factorial x == opfact x
binomial(x, y) == opbinom [x, y]
permutation(x, y) == opperm [x, y]

import F
import Kernel F

number?(x:F):Boolean ==
  if R has RetractableTo(Z) then
    ground?(x) or
    ((retractIfCan(x)@Union(Fraction(Z),"failed")) case Fraction(Z))
  else
    ground?(x)

x ** y ==
  -- Do some basic simplifications
  is?(x,POWER) =>
    args : List F := argument first kernels x
    not(#args = 2) => error "Too many arguments to **"
    number?(first args) and number?(y) =>
      oppow [first(args)**y, second args]
      oppow [first args, (second args)* y]
  -- Generic case
  exp : Union(Record(val:F,exponent:Z),"failed") := isPower x
  exp case Record(val:F,exponent:Z) =>
    expr := exp::Record(val:F,exponent:Z)
    oppow [expr.val, (expr.exponent)*y]
    oppow [x, y]

belong? op == has?(op, "comb")
fourth l == third rest l
dvpow1 l == second(l) * first(l) ** (second l - 1)
factorials x == facts(x, variables x)
factorials(x, v) == facts(x, [v])
facts(x, l) == smpfact(numer x, l) / smpfact(denom x, l)
summand l == eval(first l, retract(second l)@K, third l)

product(x:F, i:SE) ==
  dm := dummy
  opprod [eval(x, k := kernel(i)$K, dm), dm, k::F]

summation(x:F, i:SE) ==
  dm := dummy
  opsum [eval(x, k := kernel(i)$K, dm), dm, k::F]

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.

dvsum(l, x) ==
  opsum [differentiate(first l, x), second l, third l]

dvdsum(l, x) ==
  x = retract(y := third l)@SE => 0
  if member?(x, variables(h := third rest rest l)) or
    member?(x, variables(g := third rest l)) then
    error "a sum cannot be differentiated with respect to a bound"
  else
    opdsum [differentiate(first l, x), second l, y, g, h]

dvprod(l, x) ==
  dm := retract(dummy)@SE
  f := eval(first l, retract(second l)@K, dm::F)
  p := product(f, dm)
  opsum [differentiate(first l, x)/first l * p, second l, third l]

dvdprod(l, x) ==
  x = retract(y := third l)@SE => 0
  if member?(x, variables(h := third rest rest l)) or
    member?(x, variables(g := third rest l)) then
    error "a product cannot be differentiated with respect to a bound"
  else
    opdsum cons(differentiate(first l, x)/first l, rest l) * opdprod l

-- These four operations handle the conversion of sums and products to
-- OutputForm

dprod l ==
  prod(summand(l)::0, third(l)::0)

ddprod l ==
  prod(summand(l)::0, third(l)::0 = fourth(l)::0, fourth(rest l)::0)

dsum l ==
sum(summand(l)::O, third(l)::O)

ddsum l ==
  sum(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

-- The two operations handle the testing for equality of sums and products.
-- The corresponding property \verb|%specialEqual| set below is checked in
-- Kernel. Note that we can assume that the operators are equal, since this is
-- checked in Kernel itself.

equalsumprod(s1, s2) ==
  l1 := argument s1
  l2 := argument s2
  (eval(first l1, retract(second l1)@K, second l2) = first l2)

equaldsumprod(s1, s2) ==
  l1 := argument s1
  l2 := argument s2
  ((third rest l1 = third rest l2) and
   (third rest rest l1 = third rest rest l2) and
   (eval(first l1, retract(second l1)@K, second l2) = first l2))

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.

product(x:F, s:SegmentBinding F) ==
  k := kernel(variable s)$K
  dm := dummy
  opdprod [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

summation(x:F, s:SegmentBinding F) ==
  k := kernel(variable s)$K
  dm := dummy
  opdsam [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

smpfact(p, l) ==
  map(x +-> K2fact(x, l), y+->y::F, p)
  $PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)

K2fact(k, l) ==
  empty? [v for v in variables(kf := k::F) | member?(v, l)] => kf
  empty?(args:List F := [facts(a, l) for a in argument k]) => kf
  is?(k, opperm) =>
    factorial(n := first args) / factorial(n - second args)
  is?(k, opbinom) =>
    n := first args
    p := second args
    factorial(n) / (factorial(p) * factorial(n-p))
  (operator k) args
operator op ==
  is?(op, "factorial":Symbol) => opfact
  is?(op, "permutation":Symbol) => opperm
  is?(op, "binomial":Symbol) => opbinom
  is?(op, "summation":Symbol) => opsum
  is?(op, "%defsum":Symbol) => opdsum
  is?(op, "product":Symbol) => opprod
  is?(op, "%defprod":Symbol) => opdprod
  is?(op, POWER) => oppow
  error "Not a combinatorial operator"

iprod l ==
  zero? first l => 0
  one? first l => 1
  (first l = 1) => 1
  kernel(opprod, l)

isum l ==
  zero? first l => 0
  kernel(opsum, l)

idprod l ==
  member?(retract(second l)@SE, variables first l) =>
    kernel(opdprod, l)
  first(l) ** (fourth rest l - fourth l + 1)

idsum l ==
  member?(retract(second l)@SE, variables first l) =>
    kernel(opdsum, l)
  first(l) * (fourth rest l - fourth l + 1)

ifact x ==
  zero? x or one? x => 1
  zero? x or (x = 1) => 1
  kernel(opfact, x)

ibinom l ==
  n := first l
  ((p := second l) = 0) or (p = n) => 1
  one? p or (p = n - 1) => n
  (p = 1) or (p = n - 1) => n
  kernel(opbinom, l)

iperm l ==
  zero? second l => 1
  kernel(opperm, l)

if R has RetractableTo Z then
  iidsum l ==
    (ri:=retractIfCan(fourth l)@Union(Z,"failed"))
case "failed" or
  (r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
  case "failed" or
    (k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
    => idsum l
+[/eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

iidprod l ==
  (r1:=retractIfCan(fourth l)@Union(Z,"failed"))
  case "failed" or
    (r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
    case "failed" or
      (k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
      => idprod l
+[/eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

iiipow l ==
  (u := isExpt(x := first l, OPEXP)) case "failed" => kernel(oppow, l)
  rec := u::Record(var: K, exponent: Z)
  y := first argument(rec.var)
  (r := retractIfCan(y)@Union(Fraction Z, "failed")) case
    "failed" => kernel(oppow, l)
    (operator(rec.var)) (rec.exponent * y * second l)
if F has RadicalCategory then
  ipow l ==
    (r := retractIfCan(second l)@Union(Fraction Z,"failed"))
    case "failed" => iiipow l
    first(l) ** (r::Fraction(Z))
else
  ipow l ==
    (r := retractIfCan(second l)@Union(Z, "failed"))
    case "failed" => iiipow l
    first(l) ** (r::Z)
else
  ipow l ==
    zero?(x := first l) =>
      zero? second l => error "0 ** 0"
      0
    -- one? x or zero?(n := second l) => 1
      (x = 1) or zero?(n: F := second l) => 1
    -- one? n => x
      (n = 1) => x
    (u := isExpt(x, OPEXP)) case "failed" => kernel(oppow, l)
    rec := u::Record(var: K, exponent: Z)
    -- one?(y := first argument(rec.var)) or y = -1 =>
      ((y := first argument(rec.var))=1) or y = -1 =>
      (operator(rec.var)) (rec.exponent * y * n)
    kernel(oppow, l)
if R has CombinatorialFunctionCategory then
    iifact x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => ifact x
        factorial(r::R)::F

iiperm l ==
    (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
    (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
    => iperm l
    permutation(r1::R, r2::R)::F

if R has RetractableTo(Z) and F has Algebra(Fraction(Z)) then
    iibinom l ==
        (s:=retractIfCan(second l)@Union(R,"failed")) case R and
        (t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
            ans:=1::F
            for i in 0..t-1 repeat
                ans:=ans*(first l - i::R::F)
            (1/factorial t) * ans
        (s:=retractIfCan(first l-second l)@Union(R,"failed")) case R and
        (t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
            ans:=1::F
            for i in 1..t repeat
                ans:=ans*(second l+i::R::F)
            (1/factorial t) * ans
        (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
        (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
        => iibinom l
        binomial(r1::R, r2::R)::F

    -- iibinom checks those cases in which the binomial coefficient may
    -- be evaluated explicitly. Currently, the naive iterative algorithm is
    -- used to calculate the coefficient, there is room for improvement here.
else
    iibinom l ==
        (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
        (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
        => iibinom l
        binomial(r1::R, r2::R)::F
else
    iifact x == ifact x
    iibinom l == ibinom l
    iiperm l == iperm l

if R has ElementaryFunctionCategory then
    iipow l ==
        (r1:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
(r2:=retractIfCan(second l)@Union(R,"failed")) case "failed"
=> ipow l
(r1::R ** r2::R)::F
else
iipow l == ipow l
if F has ElementaryFunctionCategory then
dvpow2 l == if zero?(first l) then
  0
else
  log(first l) * first(l) ** second(l)
evaluate(opfact, iifact)@BasicOperatorFunctions1(F)
evaluate(oppow, iipow)
evaluate(opperm, iiperm)
evaluate(opbinom, iibinom)
evaluate(opsum, isum)
evaluate(opdsum, idsom)
evaluate(opprod, iprod)
evaluate(opdprod, iidprod)
derivative(oppow, [dvpow1, dvpow2])

-- These four properties define special differentiation rules for sums and
-- products.
setProperty(opsum, SPECIALDIFF, dvsum@((List F, SE) -> F) pretend None)
setProperty(opdsum, SPECIALDIFF, dvdsum@((List F, SE)->F) pretend None)
setProperty(opprod, SPECIALDIFF, dvestoprod@((List F, SE)->F) pretend None)
setProperty(opdprod, SPECIALDIFF, dvdprod@((List F, SE)->F) pretend None)

-- Set the properties for displaying sums and products and testing for
-- equality.
setProperty(opsum, SPECIALDISP, dsum@(List F -> O) pretend None)
setProperty(opdsum, SPECIALDISP, ddsum@(List F -> O) pretend None)
setProperty(opprod, SPECIALDISP, dprod@((List F) -> O) pretend None)
setProperty(opdprod, SPECIALDISP, ddprod@((List F) -> O) pretend None)
setProperty(opsum, SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
setProperty(opdsum, SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)
setProperty(opprod, SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
setProperty(opdprod, SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)

"COMBF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMBF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMBF" -> "FS"

package CDEN CommonDenominator

---

CommonDenominator examples

CommonDenominator provides functions to compute the common denominator of a finite linear aggregate of elements of the quotient field of an integral domain.

See Also:

o )show CommonDenominator
CommonDenominator (CDEN)

Exports:
  clearDenominator  commonDenominator  splitDenominator

— package CDEN CommonDenominator —

)abbrev package CDEN CommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Description:
++ CommonDenominator provides functions to compute the
++ common denominator of a finite linear aggregate of elements of
++ the quotient field of an integral domain.

CommonDenominator(R, Q, A):Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R
  A: FiniteLinearAggregate Q

Exports ==> with
  commonDenominator: A -> R
    ++ commonDenominator([q1,...,qn]) returns a common denominator
    ++ d for q1,...,qn.
  clearDenominator : A -> A
    ++ clearDenominator([q1,...,qn]) returns \spad{[p1,...,pn]} such that
    ++ \spad{qi = pi/d} where d is a common denominator for the qi’s.
  splitDenominator : A -> Record(num: A, den: R)
    ++ splitDenominator([q1,...,qn]) returns
    ++ \spad{[[p1,...,pn], d]} such that
    ++ \spad{qi = pi/d} and d is a common denominator for the qi’s.

Implementation ==> add
  clearDenominator l ==
    d := commonDenominator l
    map(x+->numer(d*x)::Q, l)
splitDenominator l ==
    d := commonDenominator l
    [map(x+->numer(d*x)::Q, l), d]

if R has GcdDomain then
    qlcm: (Q, Q) -> Q
    qlcm(a, b) == lcm(numer a, numer b)::Q
    commonDenominator l == numer reduce(qlcm, map(x+->denom(x)::Q, l), 1)
else
    commonDenominator l == numer reduce("*", map(x+->denom(x)::Q, l), 1)

package COMMONOP CommonOperators

— CommonOperators.input —

)set break resume
)sys rm -f CommonOperators.output
)spool CommonOperators.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CommonOperators
--E 1

)spool
)lisp (bye)

— CommonOperators.help —
This package exports the elementary operators, with some semantics already attached to them. The semantics that is attached here is not dependent on the set in which the operators will be applied.

See Also:
- )show CommonOperators
CHAPTER 4. CHAPTER C

Exports ==> with
  operator: Symbol -> OP
    ++ operator(s) returns an operator with name s, with the
    ++ appropriate semantics if s is known. If s is not known,
    ++ the result has no semantics.

Implementation ==> add
  dpi : List O -> 0
  dgamma : List O -> 0
  dquote : List O -> 0
  dexp : 0 -> 0
  dfact : 0 -> 0
  startUp : Boolean -> Void
  setDummyVar: (OP, NonNegativeInteger) -> OP

  brandNew?:Reference(Boolean) := ref true

  opalg := operator("rootOf":Symbol, 2)$OP
  oproot := operator("nthRoot":Symbol, 2)
  oppi := operator("pi":Symbol, 0)
  oplog := operator("log":Symbol, 1)
  oexp := operator("exp":Symbol, 1)
  opabs := operator("abs":Symbol, 1)
  opsin := operator("sin":Symbol, 1)
  opcos := operator("cos":Symbol, 1)
  optan := operator("tan":Symbol, 1)
  opcot := operator("cot":Symbol, 1)
  opsec := operator("sec":Symbol, 1)
  opcsc := operator("csc":Symbol, 1)
  opasin := operator("asin":Symbol, 1)
  opacos := operator("acos":Symbol, 1)
  opatan := operator("atan":Symbol, 1)
  opacot := operator("acot":Symbol, 1)
  opasec := operator("asec":Symbol, 1)
  opacsc := operator("acsc":Symbol, 1)
  opsinh := operator("sinh":Symbol, 1)
  opcosh := operator("cosh":Symbol, 1)
  opanh := operator("tanh":Symbol, 1)
  opcoth := operator("coth":Symbol, 1)
  opsech := operator("sech":Symbol, 1)
  opcsch := operator("csch":Symbol, 1)
  opasinh := operator("asinh":Symbol, 1)
  opacosh := operator("acosh":Symbol, 1)
  opatanh := operator("atanh":Symbol, 1)
  opacoth := operator("acoth":Symbol, 1)
opasech := operator("asech"::Symbol, 1)
opacsch := operator("acsch"::Symbol, 1)
opbox := operator("%box"::Symbol)$OP
oppren := operator("%paren"::Symbol)$OP
opquote := operator("applyQuote"::Symbol)$OP
opdiff := operator("%diff"::Symbol, 3)
opsi := operator("Si"::Symbol, 1)
opci := operator("Ci"::Symbol, 1)
oppei := operator("Ei"::Symbol, 1)
opli := operator("li"::Symbol, 1)
op erf := operator("erf"::Symbol, 1)
opli2 := operator("dilog"::Symbol, 1)
op Gamma := operator("Gamma"::Symbol, 1)
op Gamma2 := operator("Gamma2"::Symbol, 2)
op Beta := operator("Beta"::Symbol, 2)
op digamma := operator("digamma"::Symbol, 1)
oppolygamma := operator("polygamma"::Symbol, 2)
op Bessel J := operator("besselJ"::Symbol, 2)
op Bessel Y := operator("besselY"::Symbol, 2)
op Bessel I := operator("besselI"::Symbol, 2)
op Bessel K := operator("besselK"::Symbol, 2)
op Airy Ai := operator("airyAi"::Symbol, 1)
op Airy Bi := operator("airyBi"::Symbol, 1)
op int := operator("integral"::Symbol, 3)
op dint := operator("%defint"::Symbol, 5)
op fact := operator("factorial"::Symbol, 1)
opp erm := operator("permutation"::Symbol, 2)
op binom := operator("binomial"::Symbol, 2)
op pow := operator("POWER", 2)
op sum := operator("summation"::Symbol, 3)
op dsum := operator("%defsum"::Symbol, 5)
op prod := operator("product"::Symbol, 3)
op dprod := operator("%defprod"::Symbol, 5)
algop := [oproot, opalg]$List(OP)
r trigop := [opsin, opc os, optan, opcot, opsec, opcs ch, 
opasin, opacos, opatan, opcot, opasech, opacs ch]
h trigop := [opsinh, opcosh, optanh, opcoth, opsech, opcsch, 
opasinh, opcosinh, opatanh, opcoth, opasech, opacsch]
trigop := concat(r trigop, h trigop)
el emode := concat(trigop, [oppi, oplog, opexp])
primop := [opei, op li, opsi, opc i, operf, op li2, op dipl, 
opfis, op fic]
com bop := [opfact, opp erm, opbinom, opp ow, 
opsum, opdsum, opprod, opdp r od]
specop := [op Gamma, op Gamma2, op Beta, op digamma, oppolygamma, op abs, 
op Bessel J, op Bessel Y, op Bessel I, op Bessel K, op Airy Ai, 
op Airy Bi]
anyop := [oppren, opdiff, opbox, opquote]
allop := concat(concat(concat(concat(concat(
    algop, elemop), primop), combop), specop), anyop)

-- odd and even operators, must be maintained current!
evenop := [opcos, opsec, opcosh, opsech, opabs]
oddop := [opsin, opcsc, optan, opcot, opasin, opacsc, opatan,
    opsinh, opcsch, optanh, opcoth, opasinh, opacsch, opatanh, opacoth,
    opsi, operf]

-- operators whose second argument is a dummy variable
dummyvarop1 := [opdiff, opalg, opint, opsum, opprod]
-- operators whose second and third arguments are dummy variables
dummyvarop2 := [opdint, opdsum, opdprod]

operator s ==
    if (deref brandNew?) then startUp false
    for op in allop repeat
        is?(op, s) => return copy op
    operator(s)$OP

dpi l == "%pi"::Symbol::O
dfact x == postfix("!"::Symbol::O, (ATOM(x)$Lisp => x; paren x))
dquote l == prefix(quote(first(l)::O), rest l)
dgamma l == prefix(hconcat("|"::Symbol::O, overbar(" ":Symbol::O)), l)
setDummyVar(op, n) == setProperty(op, DUMMYVAR, n pretend None)
dexp x ==
e := "%e"::Symbol::O
x = 1::O => e
e ** x

fsupersub(x:List O):O == supersub("A"::Symbol::O, x)
ffac(x:List O):O == prefix(quote(first(x)$O), rest x)
ffsum(x:List O):O == sum(first x, second x, third x)
ffprod(x:List O):O == prod(first x, second x, third x)

fpren(x:List InputForm):InputForm ==
    convert concat(convert("("::Symbol@InputForm,
        concat(x, convert(")"::Symbol@InputForm)))

fpow(x:List InputForm):InputForm ==
    convert concat(convert("**"::Symbol@InputForm, x))

froot(x:List InputForm):InputForm ==
    convert [convert("**"::Symbol@InputForm, first x, 1 / second x]
startUp b ==
  brandNew?() := b
  display(opren, paren)
  display(opbox, commaSeparate)
  display(oppi, dpi)
  display(opexp, dexp)
  display(opGamma, dgamma)
  display(opGamma2, dgamma)
  display(opfact, dfact)
  display(opquote, dquote)
  display(opperm, fsupersub)
  display(opbinom, fbinomial)
  display(oppow, fpower)
  display(opsum, fsum)
  display(opprod, fprod)
  display(opint, fint)
  input(opren, fpren)
  input(oppow, fpow)
  input(oproot, froot)
  for op in algop repeat assert(op, ALGOP)
  for op in rtrigop repeat assert(op, "rtrig")
  for op in htrigop repeat assert(op, "htrig")
  for op in trigop repeat assert(op, "trig")
  for op in elemop repeat assert(op, "elem")
  for op in primop repeat assert(op, "prim")
  for op in combop repeat assert(op, "comb")
  for op in specop repeat assert(op, "special")
  for op in anyop repeat assert(op, "any")
  for op in evenop repeat assert(op, EVEN)
  for op in oddop repeat assert(op, ODD)
  for op in dummyvarop1 repeat setDummyVar(op, 1)
  for op in dummyvarop2 repeat setDummyVar(op, 2)
  assert(opren, "linear")
  void

---

 COMMONOP.dotabb ---

"COMMONOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMMONOP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"COMMONOP" -> "ALIST"

---
package COMMUPC CommuteUnivariatePolynomialCategory

--- CommuteUnivariatePolynomialCategory.input ---

)set break resume
)sys rm -f CommuteUnivariatePolynomialCategory.output
)spool CommuteUnivariatePolynomialCategory.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CommuteUnivariatePolynomialCategory
--E 1

)spool
)lisp (bye)

---

--- CommuteUnivariatePolynomialCategory.help ---

====================================================================
CommuteUnivariatePolynomialCategory examples
====================================================================

A package for swapping the order of two variables in a tower of two
UnivariatePolynomialCategory extensions.

See Also:
  o )show CommuteUnivariatePolynomialCategory
CommuteUnivariatePolynomialCategory (COMMUPC)

Exports:

---package COMMUPC CommuteUnivariatePolynomialCategory---

)abbrev package COMMUPC CommuteUnivariatePolynomialCategory
++ Author: Manuel Bronstein
++ Description:
++ A package for swapping the order of two variables in a tower of two
++ UnivariatePolynomialCategory extensions.

CommuteUnivariatePolynomialCategory(R, UP, UPUP): Exports == Impl where
R : Ring
UP : UnivariatePolynomialCategory R
UPUP: UnivariatePolynomialCategory UP

N ==> NonNegativeInteger

Exports ==> with
  swap: UPUP -> UPUP
  ++ swap(p(x,y)) returns p(y,x).

Impl ==> add
  makePoly: (UP, N) -> UPUP
  -- converts P(x,y) to P(y,x)
  swap poly ==
    ans:UPUP := 0
    while poly ^= 0 repeat
      poly := reductum poly
      ans := ans + makePoly(leadingCoefficient poly, degree poly)
    ans
  makePoly(poly, d) ==
    ans:UPUP := 0
while poly ^= 0 repeat
  ans := ans +
  monomial(monomial(leadingCoefficient poly, d), degree poly)
  poly := reductum poly
ans

— COMMUPC.dotabb —

"COMMUPC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMMUPC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"COMMUPC" -> "PFECAT"

— package COMPFACT ComplexFactorization —

— ComplexFactorization.input —

)set break resume
)sys rm -f ComplexFactorization.output
)spool ComplexFactorization.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ComplexFactorization
--E 1

)spool
)lisp (bye)

— ComplexFactorization.help —

====================================================================
ComplexFactorization examples
====================================================================

This package has no description
See Also:
o )show ComplexFactorization

ComplexFactorization (COMPFACT)

Exports:
factor

— package COMPFACT ComplexFactorization —

)abbrev package COMPFACT ComplexFactorization
++ Description:
++ This package has no description

ComplexFactorization(RR,PR) : C == T where
RR : EuclideanDomain  -- R is Z or Q
PR : UnivariatePolynomialCategory Complex RR
R  ==> Complex RR
I  ==> Integer
RN ==> Fraction I
GI ==> Complex I
GRN ==> Complex RN

C == with

factor : PR -> Factored PR
  ++ factor(p) factorizes the polynomial p with complex coefficients.

T == add
CHAPTER 4. CHAPTER C

SUP ==> SparseUnivariatePolynomial  
fUnion ==> Union("nil", "sqfr", "irred", "prime")  
FF ==> Record(flg:fUnion, fctr:PR, xpnt:Integer)  
SAEF := SimpleAlgebraicExtensionAlgFactor(SUP RN,GRN,SUP GRN)  
UPCF2 := UnivariatePolynomialCategoryFunctions2(R,PR,GRN,SUP GRN)  
UPCFB := UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,R,PR)  

myMap(r:R) : GRN ==  
  R is GI =>  
    cr :GI := r pretend GI  
    complex((real cr)::RN,(imag cr)::RN)  
  R is GRN => r pretend GRN

compND(cc:GRN):Record(cnum:GI,cden:Integer) ==  
  ccr:=real cc  
  cci:=imag cc  
  dccr:=denom ccr  
  dcci:=denom cci  
  ccd:=lcm(dccr,dcci)  
  [complex(((ccd exquo dccr)::Integer)*numer ccr,  
            ((ccd exquo dcci)::Integer)*numer cci),ccd]

conv(f:SUP GRN) :Record(convP:SUP GI, convD:RN) ==  
  pris:SUP GI :=0  
  dris:Integer:=1  
  dris1:Integer:=1  
  pdris:Integer:=1  
  for i in 0..(degree f) repeat  
    (cf:= coefficient(f,i)) = 0 => "next i"  
    cdf:=compND cf  
    dris:=lcm(cdf.cden,dris1)  
    pris:=(((dris exquo dris1)::Integer)*pris +  
              ((dris exquo cdf.cden)::Integer)*  
              monomial(cdf.cnum,i)$(SUP GI)  
    dris1:=dris  
  [pris,dris::RN]

backConv(ffr:Factored SUP GRN) : Factored PR ==  
  R is GRN =>  
    makeFR((unit ffr) pretend PR,[[f.flg,(f.fctr) pretend PR,f.xpnt]  
                             for f in factorList ffr])  
  R is GI =>  
    const:=unit ffr  
    ris: List FF :=[]  
    for ff in factorList ffr repeat  
      fact:=primitivePart(conv(ff.fctr).convP)  
      expf:=ff.xpnt  
      ris:=cons([[ff.flg,fact pretend PR,expf],ris)  
    lc:GRN := myMap leadingCoefficient(fact pretend PR)  
    const:= const*(leadingCoefficient(ff.fctr)/lc)**expf
uconst:GI := compND(coefficient(const,0)).cnum
makeFR((uconst pretend R)::PR,ris)

factor(pol : PR) : Factored PR ==
ratPol: SUP GRN := 0
ratPol:=map(myMap, pol)$UPCF2
ffr:=factor ratPol
backConv ffr
ComplexFunctions2 examples
====================================================================

This package extends maps from underlying rings to maps between complex over those rings.

See Also:
o )show ComplexFunctions2

ComplexFunctions2 (COMPLEX2)

Exports:
map

| package COMPLEX2 ComplexFunctions2 |

)abbrev package COMPLEX2 ComplexFunctions2
++ Description:
++ This package extends maps from underlying rings to maps between complex over those rings.

ComplexFunctions2(R:CommutativeRing, S:CommutativeRing): with
  map: (R -> S, Complex R) -> Complex S
    ++ map(f,u) maps f onto real and imaginary parts of u.
  == add
    map(fn, gr) == complex(fn real gr, fn imag gr)

— COMPLEX2.dotabb —
package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation

-------------------
ComplexIntegerSolveLinearPolynomialEquation.help

This package provides the generalized euclidean algorithm which is needed as the basic step for factoring polynomials.

See Also:
  o )show ComplexIntegerSolveLinearPolynomialEquation

-------------------
ComplexIntegerSolveLinearPolynomialEquation.examples

---

ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE)

Exports:
solveLinearPolynomialEquation

— package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation —

)abbrev package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation
++ Author: James Davenport
++ Date Created: 1990
++ Description:
++ This package provides the generalized euclidean algorithm which is
++ needed as the basic step for factoring polynomials.

ComplexIntegerSolveLinearPolynomialEquation(R,CR): C == T
where
  CP ==> SparseUnivariatePolynomial CR
  R:IntegerNumberSystem
  CR:ComplexCategory(R)
  C == with
    solveLinearPolynomialEquation: (List CP,CP) -> Union(List CP,"failed")
    ++ solveLinearPolynomialEquation([f1, ..., fn], g)
    ++ where (fi relatively prime to each other)
    ++ returns a list of ai such that
    ++ g = sum ai prod fj (j \= i) or
    ++ equivalently g/prod fj = sum (ai/fi)
    ++ or returns "failed" if no such list exists

T == add
oldlp:List CP := []
slpePrime:R:=(2::R)
oldtable:Vector List CP := empty()
solveLinearPolynomialEquation(lp,p) ==
  if (oldlp ^= lp) then
    -- we have to generate a new table
    deg:= _+[/degree u for u in lp]
    ans:Union(Vector List CP,"failed"):="failed"
slpePrime::=67108859::R -- 2**26 -5 : a prime
  -- a good test case for this package is
  -- (good question?)
while (ans case "failed") repeat
  ans:=tablePow(deg,complex(slpePrime,0),lp)$GenExEuclid(CR,CP)
  if (ans case "failed") then
    slpePrime:= slpePrime-4::R
    while not prime?(slpePrime)$IntegerPrimesPackage(R) repeat
      slpePrime:= slpePrime-4::R
  end
  oldtable:=(ans:: Vector List CP)
answer:=solveid(p,complex(slpePrime,0),oldtable)
answer

———

— CINTSLPE.dotabb —

"CINTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CINTSLPE"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CINTSLPE" -> "COMPCAT"

———

package COMPLPAT ComplexPattern

— ComplexPattern.input —

)set break resume
)sys rm -f ComplexPattern.output
)spool ComplexPattern.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexPattern
--E 1

)spool
)lisp (bye)

———

— ComplexPattern.help —
ComplexPattern examples

This package supports converting complex expressions to patterns

See Also:
- )show ComplexPattern
ipat : Pattern R := patternVariable("%i":Symbol, true, false, false)

convert(cs) ==
  zero? imag cs => convert real cs
  convert real cs + ipat * convert imag cs

package CPMATCH ComplexPatternMatch

— ComplexPatternMatch.input —

)set break resume
)sys rm -f ComplexPatternMatch.output
)spool ComplexPatternMatch.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexPatternMatch
--E 1

)spool
)lisp (bye)

— ComplexPatternMatch.help —

====================================================================
ComplexPatternMatch examples
====================================================================
This package supports matching patterns involving complex expressions

See Also:
o )show ComplexPatternMatch

ComplexPatternMatch (CPMATCH)

Exports:
patternMatch

— package CPMATCH ComplexPatternMatch —

)abbrev package CPMATCH ComplexPatternMatch
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Description:
++ This package supports matching patterns involving complex expressions

ComplexPatternMatch(R, S, CS) : C == T where
R: SetCategory
S: Join(PatternMatchable R, CommutativeRing)
CS: ComplexCategory S
PMRS ==> PatternMatchResult(R, CS)
PS ==> Polynomial S
C == with
  if PS has PatternMatchable(R) then
    patternMatch: (CS, Pattern R, PMRS) -> PMRS
    ++ patternMatch(cexpr, pat, res) matches the pattern pat to the
    ++ complex expression cexpr. res contains the variables of pat
    ++ which are already matched and their matches.
import PatternMatchPushDown(R, S, CS)
import PatternMatchResultFunctions2(R, PS, CS)
import PatternMatchResultFunctions2(R, CS, PS)
ivar : PS := "%i":Symbol::PS

makeComplex(p:PS):CS ==
  up := univariate p
  degree up > 1 => error "not linear in %i"
  icoef:=leadingCoefficient(up)
  rcoef:=leadingCoefficient(reductum p)
  complex(rcoef,icoef)

makePoly(cs:CS):PS == real(cs)*ivar + imag(cs)::PS

if PS has PatternMatchable(R) then
  patternMatch(cs, pat, result) ==
    zero? imag cs =>
      patternMatch(real cs, pat, result)
    map(makeComplex,
      patternMatch(makePoly cs, pat, map(makePoly, result)))

package CRFP ComplexRootFindingPackage

"CPMATCH"
"COMPCAT"
"CPMATCH" -> "COMPCAT"
ComplexRootFindingPackage provides functions to find all roots of a polynomial \( p \) over the complex number by using Plesken's idea to calculate in the polynomial ring modulo \( f \) and employing the Chinese Remainder Theorem.

In this first version, the precision (see digits) is not increased when this is necessary to avoid rounding errors. Hence it is the user's responsibility to increase the precision if necessary.

Note also, if this package is called with e.g. Fraction Integer, the precise calculations could require a lot of time.

Also note that evaluating the zeros is not necessarily a good check whether the result is correct: already evaluation can cause rounding errors.

See Also:
- \( \)show ComplexRootFindingPackage

---

ComplexRootFindingPackage.help

---

ComplexRootFindingPackage examples

---

ComplexRootFindingPackage provides functions to find all roots of a polynomial \( p \) over the complex number by using Plesken's idea to calculate in the polynomial ring modulo \( f \) and employing the Chinese Remainder Theorem.

In this first version, the precision (see digits) is not increased when this is necessary to avoid rounding errors. Hence it is the user's responsibility to increase the precision if necessary.

Note also, if this package is called with e.g. Fraction Integer, the precise calculations could require a lot of time.

Also note that evaluating the zeros is not necessarily a good check whether the result is correct: already evaluation can cause rounding errors.

See Also:
- \( \)show ComplexRootFindingPackage

---
ComplexRootFindingPackage (CRFP)

Exports:
complexZeros divisorCascade factor graeffe norm
pleskenSplit reciprocalPolynomial rootRadius schwerpunkt setErrorBound
startPolynomial

— package CRFP ComplexRootFindingPackage —

)abbrev package CRFP ComplexRootFindingPackage
++ Author: J. Grabmeier
++ Date Created: 31 January 1991
++ Date Last Updated: 12 April 1991
++ References: J. Grabmeier: On Plesken's root finding algorithm,
++ in preparation
++ A. Schoenhage: The fundamental theorem of algebra in terms of computational
++ complexity, preliminary report, Univ. Tuebingen, 1982
++ Description:
++ \spadtype{ComplexRootFindingPackage} provides functions to
++ find all roots of a polynomial \( p \) over the complex number by
++ using Plesken's idea to calculate in the polynomial ring
++ modulo \( f \) and employing the Chinese Remainder Theorem.
++ In this first version, the precision (see digits)
++ is not increased when this is necessary to
++ avoid rounding errors. Hence it is the user's responsibility to
++ increase the precision if necessary.
++ Note also, if this package is called with e.g. \spadtype{Fraction Integer},
++ the precise calculations could require a lot of time.
++ Also note that evaluating the zeros is not necessarily a good check
++ whether the result is correct: already evaluation can cause
++ rounding errors.

ComplexRootFindingPackage(R, UP): public == private where
-- R : Join(Field, OrderedRing, CharacteristicZero)
-- Float not in CharacteristicZero !|
R : Join(Field, OrderedRing)
UP : UnivariatePolynomialCategory Complex R
CHAPTER 4. CHAPTER C

C ==> Complex R
FR ==> Factored
I ==> Integer
L ==> List
FAE ==> Record(factors : L UP, error : R)
NNI ==> NonNegativeInteger
OF ==> OutputForm
ICF ==> IntegerCombinatoricFunctions(I)

public ==> with
complexZeros : UP -> L C
++ complexZeros(p) tries to determine all complex zeros
++ of the polynomial p with accuracy given by the package
++ constant globalEps which you may change by setErrorBound.
complexZeros : (UP, R) -> L C
++ complexZeros(p, eps) tries to determine all complex zeros
++ of the polynomial p with accuracy given by eps.
divisorCascade : (UP,UP, Boolean) -> L FAE
++ divisorCascade(p,tp) assumes that degree of polynomial tp
++ is smaller than degree of polynomial p, both monic.
++ A sequence of divisions are calculated
++ using the remainder, made monic, as divisor
++ for the the next division. The result contains also the error of the
++ factorizations, i.e. the norm of the remainder polynomial.
++ If info is true, then information messages are issued.
divisorCascade : (UP,UP) -> L FAE
++ divisorCascade(p,tp) assumes that degree of polynomial tp
++ is smaller than degree of polynomial p, both monic.
++ A sequence of divisions is calculated
++ using the remainder, made monic, as divisor
++ for the the next division. The result contains also the error of the
++ factorizations, i.e. the norm of the remainder polynomial.
factor: (UP,R,Boolean) -> FR UP
++ factor(p, eps, info) tries to factor p into linear factors
++ with error atmost eps. An overall error bound
++ eps0 is determined and iterated tree-like calls
++ to pleskenSplit are used to get the factorization.
++ If info is true, then information messages are given.
factor: (UP,R) -> FR UP
++ factor(p, eps) tries to factor p into linear factors
++ with error atmost eps. An overall error bound
++ eps0 is determined and iterated tree-like calls
++ to pleskenSplit are used to get the factorization.
factor: UP -> FR UP
++ factor(p) tries to factor p into linear factors
++ with error atmost globalEps, the internal error bound,
++ which can be set by setErrorBound. An overall error bound
++ eps0 is determined and iterated tree-like calls
++ to pleskenSplit are used to get the factorization.
graeffe : UP -> UP
++ graeffe p determines q such that \spad{q(-z**2) = p(z)*p(-z)}.
++ Note that the roots of q are the squares of the roots of p.
norm : UP -> R
++ norm(p) determines sum of absolute values of coefficients
++ Note that this function depends on abs.
pleskenSplit: (UP, R, Boolean) -> FR UP
++ pleskenSplit(poly,eps,info) determines a start polynomial start
++ by using "startPolynomial then it increases the exponent
++ n of start ** n mod poly to get an approximate factor of
++ poly, in general of degree "degree poly -1". Then a divisor
++ cascade is calculated and the best splitting is chosen, as soon
++ as the error is small enough.
---++ In a later version we plan
---++ to use the whole information to get a split into more than 2
---++ factors.
++ If info is true, then information messages are issued.
pleskenSplit: (UP, R) -> FR UP
++ pleskenSplit(poly, eps) determines a start polynomial start
++ by using "startPolynomial then it increases the exponent
++ n of start ** n mod poly to get an approximate factor of
++ poly, in general of degree "degree poly -1". Then a divisor
++ cascade is calculated and the best splitting is chosen, as soon
++ as the error is small enough.
---++ In a later version we plan
---++ to use the whole information to get a split into more than 2
---++ factors.
reciprocalPolynomial: UP -> UP
++ reciprocalPolynomial(p) calculates a polynomial which has exactly
++ the inverses of the non-zero roots of p as roots, and the same
++ number of 0-roots.
rootRadius: (UP,R) -> R
++ rootRadius(p,errQuot) calculates the root radius of p with a
++ maximal error quotient of errQuot.
rootRadius: UP -> R
++ rootRadius(p) calculates the root radius of p with a
++ maximal error quotient of 1+globalEps, where
++ globalEps is the internal error bound, which can be
++ set by setErrorBound.
schwerpunkt: UP -> C
++ schwerpunkt(p) determines the 'Schwerpunkt' of the roots of the
++ polynomial p of degree n, i.e. the center of gravity, which is
++ coefficient of \spad{x**(n-1)} divided by
++ n times coefficient of \spad{x**n}.
setErrorBound : R -> R
++ setErrorBound(eps) changes the internal error bound,
-- by default being 10 ** (-20) to eps, if R is
++ by default being 10 ** (-3) to eps, if R is
++ a member in the category \spadtype{QuotientFieldCategory Integer}.
++ The internal globalDigits is set to
++ \em ceiling(1/r)**2*10 being 10**7 by default.
startPolynomial: UP -> Record(start: UP, factors: FR UP)
++ startPolynomial(p) uses the ideas of Schoenhage's
++ variant of Graeffe's method to construct circles which separate
++ roots to get a good start polynomial, i.e. one whose
++ image under the Chinese Remainder Isomorphism has both entries
++ of norm smaller and greater or equal to 1. In case the
++ roots are found during internal calculations.
++ The corresponding factors
++ are in factors which are otherwise 1.

private ==> add

Rep := ModMonic(C, UP)

-- constants
c : C
r : R
--globalDigits : I := 10 ** 41
globalDigits : I := 10 ** 7
globalEps : R :=
  -a : R := (100000000000000000000000000000000 :: I) :: R
  a : R := (1000 :: I) :: R
  1/a
emptyLine : OF := " "
dashes : OF := center "---------------------------------------------------"
dots : OF := center "...................................................
one : R := 1*R
two : R := 2 * one
ten : R := 10 * one
eleven : R := 11 * one
weakEps := eleven/ten
--invLog2 : R := 1/log10 (2*one)

-- signatures of local functions
absC : C -> R
--
absR : R -> R
--
calculateScale : UP -> R
--
makeMonic : UP -> UP
-- 'makeMonic p' divides 'p' by the leading coefficient,
-- to guarantee new leading coefficient to be 1*R we cannot
-- simply divide the leading monomial by the leading coefficient
-- because of possible rounding errors
min: (FAE, FAE) -> FAE
-- takes factorization with smaller error
nthRoot : (R, NNI) -> R
-- nthRoot(r,n) determines an approximation to the n-th
-- root of r, if \spadtype{R} has ?**?: (R,Fraction Integer)->R
-- we use this, otherwise we use approxNthRoot via
-- \spadtype{Integer}

shift: (UP,C) -> UP
-- shift(p,c) changes p(x) into p(x+c), thereby modifying the
-- roots u_j of p to the roots (u_j - c) of shift(p,c)

scale: (UP,C) -> UP
-- scale(p,c) changes p(x) into p(cx), thereby modifying the
-- roots u_j of p to the roots ((1/c) u_j) of scale(p,c)

-- implementation of exported functions

complexZeros(p,eps) ==
--r1 : R := rootRadius(p,weakEps)
--eps0 : R = r1 * nthRoot(eps, degree p)
-- right now we are content with
eps0 : R := eps/(ten ** degree p)
facs : FR UP := factor(p,eps0)
[-coefficient(linfac.factor,0) for linfac in factors facs]

complexZeros p == complexZeros(p,globalEps)

setErrorBound r ==
r <= 0 => error "setErrorBound: need error bound greater 0"
globalEps := r
if R has QuotientFieldCategory Integer then
  rd : Integer := ceiling(1/r)
globalDigits := rd * rd * 10
lof : List OF := _
  ["setErrorBound: internal digits set to",globalDigits::OF]
print hconcat lof
messagePrint "setErrorBound: internal error bound set to"
globalEps

pleskenSplit(poly,eps,info) ==
p := makeMonic poly
fp : FR UP
if not zero? (md := minimumDegree p) then
  fp : FR UP := irreducibleFactor(monomial(1,1)$UP,md)$(FR UP)
p := p quo monomial(1,md)$UP
sP : Record(start: UP, factors: FR UP) := startPolynomial p
fp : FR UP := sP.factors
-- if not one? fp then
if not (fp = 1) then
  qr : Record(quotient: UP, remainder: UP):= divide(p,makeMonic expand fp)
p := qr.quotient
st := sP.start
zero? degree st => fp
   -- we calculate in ModMonic(C, UP),
   -- next line defines the polynomial, which is used for reducing
setPoly p
nm : R := eps
split : FAE
sR : Rep := st :: Rep
psR : Rep := sR ** (degree poly)

notFoundSplit : Boolean := true
-- if info then
--   lof : L OF := ["not successfull, new exponent:", nn::OF]
--   print hconcat lof
psR := psR * psR * sR -- exponent (2*d +1)
   -- be careful, too large exponent results in rounding errors
   -- tp is the first approximation of a divisor of poly:
   tp : UP := lift psR
zero? degree tp =>
   if info then print "we leave as we got constant factor"
     nilFactor(poly,1)$(FR UP)
   -- this was the case where we don't find a non-trivial factorization
   -- we refine tp by repeated polynomial division and hope that
   -- the norm of the remainder gets small from time to time
splits : L FAE := divisorCascade(p, makeMonic tp, info)
split := reduce(min, splits)
notFoundSplit := (eps <= split.error)

for fac in split.factors repeat
  fp :=
  -- one? degree fac => fp * nilFactor(fac,1)$(FR UP)
  (degree fac = 1) => fp * nilFactor(fac,1)$(FR UP)
  fp * irreducibleFactor(fac,1)$(FR UP)
fp

startPolynomial p == -- assume minimumDegree is 0
   --print (p :: OF)
   fp : FR UP := 1
   -- one? degree p =>
   (degree p = 1) =>
     p := makeMonic p
     [p, irreducibleFactor(p,1)]
startPoly : UP := monomial(1,1)$UP
eps : R := weakEps -- 10 per cent errors allowed
r1 : R := rootRadius(p, eps)
rd : R := 1/rootRadius(reciprocalPolynomial p, eps)
(r1 > (2::R)) and (rd < 1/(2::R)) => [startPoly, fp] -- unit circle splitting!
-- otherwise the norms of the roots are too closed so we
-- take the center of gravity as new origin:
u : C := schwerpunkt p
\[
\text{startPoly} := \text{startPoly-monomial}(u,0)
\]

\[
p := \text{shift}(p,-u)
\]

-- determine new rootRadius:

\[
r1 : R := \text{rootRadius}(p, \epsilon)
\]

\[
\text{startPoly} := \text{startPoly}/(r1::C)
\]

-- use one of the 4 points ri*zeta, where zeta is a 4th root of unity
-- as new origin, this could be changed to an arbitrary list
-- of elements of norm 1.

\[
\text{listOfCenters} : L C := \{\text{complex}(r1,0), \text{complex}(0,r1), \_
\text{complex}(-r1,0), \text{complex}(0,-r1)\}
\]

\[
lp : L \text{UP} := \{\text{shift}(p,v) \text{ for } v \text{ in listOfCenters}\}
\]

-- next we check if one of these centers is a root

\[
\text{centerIsRoot} : \text{Boolean} := \text{false}
\]

for i in 1..\text{maxIndex} lp repeat

\[
\text{if } (\text{mD} := \text{minimumDegree } lp\text{.i}) > 0 \text{ then}
\]

\[
\text{pp} : \text{UP} := \text{monomial}(1,1)-\text{monomial}(\text{listOfCenters}\text{.i}-u,0)
\]

\[
\text{centerIsRoot} := \text{true}
\]

\[
\text{fp} := \text{fp} * \text{irreducibleFactor}(\text{pp},\text{mD})
\]

\[
\text{centerIsRoot} \Rightarrow
\]

\[
p := \text{shift}(p,u) \text{ quo expand fp}
\]

-- print (p::OF)

\[
\text{zero? degree } p \Rightarrow \{p,fp\}
\]

\[
\text{sP} := \text{startPolynomial}(p)
\]

\[
\{\text{sP.start,fp}\}
\]

-- choose the best one w.r.t. maximal quotient of norm of largest
-- root and norm of smallest root

\[
lpr1 : L R := \{\text{rootRadius}(q,\epsilon) \text{ for } q \text{ in lp}\}
\]

\[
lprd : L R := \{1/\text{rootRadius}(\text{reciprocalPolynomial} q,\epsilon) \text{ for } q \text{ in lp}\}
\]

-- later we should check here if an rd is smaller than globalEps

\[
lq : L R := \{
\}
\]

for i in 1..\text{maxIndex} lpr1 repeat

\[
lq := \text{cons}(lpr1\text{.i}/lprd\text{.i}, lq)
\]

-- lq : L R := [(i/s)::R for i in lpr1 for s in lprd]

\[
lq := \text{reverse } lq
\]

\[
\text{po} := \text{position}(\text{reduce}(\text{max},lq),lq)
\]

-- p := lp\text{.po}

\[
-- lrr : L R := [\text{rootRadius}(p,i,1+\epsilon) \text{ for } i \text{ in } 2..(\text{degree}(p)-1)]
\]

\[
-- lrr := \text{concat}(\text{concat}(lpr1\text{.po},lrr),lprd\text{.po})
\]

\[
-- lu : L R := [(lrr\text{.i} + lrr\text{.i+1})/2, lrr\text{.i} for i \text{ in } 1..(\text{maxIndex}(lrr)-1)]
\]

\[
\text{startPoly} - \text{monomial}(\text{listOfCenters}\text{.po},0),\text{fp}
\]

\[
norm p ==
\]

-- reduce(_+$R,\text{map}(\text{absC},\text{coefficients } p))

\[
nm : R := 0
\]

for c in \text{coefficients } p repeat

\[
nm := nm + \text{absC} c
\]

nm

\[
\text{pleskenSplit}(\text{poly},\epsilon) = \text{pleskenSplit}(\text{poly},\epsilon,\text{false})
\]
CHAPTER 4. CHAPTER C

graeffe p ==
-- If \( p = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n \)
-- and \( q = b_0 x^n + b_1 x^{n-1} + \ldots + b_{n-1} x + b_n \)
-- are such that \( q(-x^2) = p(x)p(-x) \), then
-- \( b_k := a_k^2 + 2 \sum_{l=1}^{\min(k, n-k)} (-1)^l a_l a_{l+k} \)
-- graeffe(p) constructs q using these identities.

n : NNI := degree p
aForth : L C := []
for k in 0..n repeat -- aForth = [a0, a1, ..., a_{n-1}, an]
aForth := cons(coefficient(p, k::NNI), aForth)
aBack : L C := [] -- after k steps
-- aBack = [a_k, a_{k-1}, ..., a1, a0]
gp : UP := 0$UP
for k in 0..n repeat
ak : C := first aForth
aForth := rest aForth
aForthCopy : L C := aForth -- we iterate over aForth and
aBackCopy : L C := aBack -- aBack but do not want to
-- destroy them
sum : C := 0
const : I := -1 -- after i steps const = (-1)**i
for aminus in aBack for aplus in aForth repeat
-- after i steps aminus = a_{k-i} and aplus = a_{k+i}
sum := sum + const * aminus * aplus
aForthCopy := rest aForthCopy
aBackCopy := rest aBackCopy
const := -const
gp := gp + monomial(ak*ak + 2 * sum, (n-k)::NNI)
aBack := cons(ak, aBack)
gp

rootRadius(p, errorQuotient) ==
errorQuotient <= 1$R =>
    error "rootRadius: second Parameter must be greater than 1"
pp : UP := p
rho : R := calculateScale makeMonic pp
rR : R := rho
pp := makeMonic scale(pp, complex(rho, 0$R))
expo : NNI := 1
d : NNI := degree p
currentError : R := nthRoot(2::R, 2)
currentError := d*20*currentError
while nthRoot(currentError, expo) >= errorQuotient repeat
    -- if info then print (expo :: OF)
    pp := graeffe pp
    rho := calculateScale pp
    expo := 2 * expo

PACKAGE CRFP COMPLEXROOTFINDINGPACKAGE

\[ rR := \text{nthRoot}(\rho, \text{expo}) \times rR \]
\[ \text{pp := makeMonic scale(pp, complex(\rho, 0R))} \]
\[ rR \]

\[ \text{rootRadius}(p) = \text{rootRadius}(p, 1 + \text{globalEps}) \]

\[ \text{schwerpunkt } p = \]
zero? p \Rightarrow 0C
zero? (d := \text{degree} p) \Rightarrow \text{error} _
"\text{schwerpunkt: non-zero const. polynomial has no roots and no schwerpunkt}"
-- coefficient of \( x^d \) and \( x^{(d-1)} \)
IC : C := \text{coefficient}(p, d) -- ^= 0
nC : C := \text{coefficient}(p, (d-1) \text{ pretend NNI})
(denom := \text{recip} ((d::I::C)*1C)) \text{ case "failed" } \Rightarrow \text{error} "\text{schwerpunkt: }_
\text{degree} \times \text{leadingCoefficient not invertible in ring of coefficients"
- (nC*denom::C))

\[ \text{reciprocalPolynomial } p = \]
zero? p \Rightarrow 0
\[ d : \text{NNI} := \text{degree} p \]
\[ m : \text{NNI} := d + \text{minimumDegree} p \]
\[ lm : \text{L UP} := \text{[monomial(coefficient(p,i),(md-i) :: NNI) for i in 0..d]} \]
\[ \text{sol := reduce(\_+, lm)} \]

\[ \text{divisorCascade}(p, \text{tp}, \text{info}) = \]
\[ \text{lfae : L FAE := nil()} \]
for i in 1..\text{degree} tp while (\text{degree} tp > 0) repeat
-- USE monicDivide !!!
\[ qr : \text{Record}(\text{quotient: UP, remainder: UP}) := \text{divide}(p, tp) \]
\[ \text{factor1 : UP := tp} \]
\[ \text{factor2 : UP := makeMonic qr.quotient} \]
-- refinement of tp:
\[ \text{tp := qr.remainder} \]
\[ \text{nm : R := norm \text{tp}} \]
\[ \text{listOfFactors : L UP := cons(factor2,nil())$(L \text{UP})} \]
\[ \text{listOfFactors := cons(factor1, listOfFactors)} \]
\[ \text{lfae := cons([listOfFactors,nm], lfae)} \]
if info then
\[ --\text{lof : L OF := [i :: OF,"-th division:"::OF]} \]
--print center box hconcat lof
print emptyLine
\[ \text{lof : L OF := ["error polynomial has degree " ::OF, _
(\text{degree} \text{tp})::OF, " and norm " :: OF, \text{nm} :: OF]} \]
print center hconcat lof
\[ \text{lof : L OF := ["degrees of factors:" ::OF, _
(\text{degree factor1})::OF, "", (\text{degree factor2})::OF]} \]
print center hconcat lof
if info then print emptyLine
reverse lfae
divisorCascade(p, tp) == divisorCascade(p, tp, false)

factor(poly, eps) == factor(poly, eps, false)
factor(p) == factor(p, globalEps)

factor(poly, eps, info) ==
result : FR UP := coerce monomial(leadingCoefficient poly, 0)
d : NNI := degree poly
-- should be
-- den : R := (d::I):R * two**(d::Integer) * norm poly
-- eps0 : R := eps / den
-- for now only
eps0 : R := eps / (ten*ten)
-- one? d => irreducibleFactor(poly, 1)$(FR UP)
(d = 1) => irreducibleFactor(poly, 1)$(FR UP)
listOfFactors : L Record(factor: UP, exponent: I) := list [makeMonic poly, 1]
if info then
lof : L OF := [dashes, dots, "list of Factors:", dots, listOfFactors::OF, dashes, "list of Linear Factors:", dots, result::OF, dashes, dashes]
print vconcat lof
while not null listOfFactors repeat
p : UP := (first listOfFactors).factor
exponentOfp : I := (first listOfFactors).exponent
listOfFactors := rest listOfFactors
if info then
lof : L OF := ["just now we try to split the polynomial:", p::OF]
print vconcat lof
split : FR UP := pleskenSplit(p, eps0, info)
-- one? numberOfFactors split =>
(numberOfFactors split = 1) =>
-- in a later version we will change error bound and
-- accuracy here to deal this case as well
lof : L OF := ["factor: couldn't split factor", center(p :: OF), "with required error bound"]
print vconcat lof
result := result * nilFactor(p, exponentOfp)
-- now we got 2 good factors of p, we drop p and continue
-- with the factors, if they are not linear, or put a
-- linear factor to the result
for rec in factors(split)$(FR UP) repeat
newFactor : UP := rec.factor
expOfFactor := exponentOfp * rec.exponent
-- one? degree newFactor =>
(degree newFactor = 1) =>
result := result * nilFactor(newFactor, expOfFactor)
listOfFactors := cons([newFactor, expOfFactor], listOfFactors)
result
-- implementation of local functions

absC c == nthRoot(norm(c)$C,2)

absR r ==
  r < 0 => -r
  r

min(fae1,fae2) ==
  fae2.error < fae1.error => fae2
  fae1

calculateScale p ==
  d := degree p
  maxi := 0
  for j in 1..d for cof in rest coefficients p repeat
    -- here we need abs: R -> R
    rc : R := absR real cof
    ic : R := absR imag cof
    locmax: R := max(rc,ic)
    maxi := max( nthRoot( locmax/(binomial(d,j)$ICF::R), j), maxi)
    -- Maybe I should use some type of logarithm for the following:
    maxi = 0$R => error("Internal Error: scale cannot be 0")
    rho : R := one
    rho < maxi =>
      while rho < maxi repeat rho := ten * rho
      rho / ten
    while maxi < rho repeat rho := rho / ten
    rho = 0 => one
    rho

makeMonic p ==
  p = 0 => p
  monomial(1,degree p)$UP + (reductum p)/(leadingCoefficient p)

scale(p, c) ==
  -- eval(p,cx) is missing !!
  eq : Equation UP := equation(monomial(1,1), monomial(c,1))
  eval(p,eq)
  -- improvement?: direct calculation of the new coefficients

shift(p,c) ==
  rhs : UP := monomial(1,1) + monomial(c,0)
  eq : Equation UP := equation(monomial(1,1), rhs)
  eval(p,eq)
  -- improvement?: direct calculation of the new coefficients

nthRoot(r,n) ==
  R has RealNumberSystem => r ** (1/n)
  R has QuotientFieldCategory Integer =>
  den : I := approxNthRoot(globalDigits * denom r ,n)$IntegerRoots(I)
  num : I := approxNthRoot(globalDigits * numer r ,n)$IntegerRoots(I)
  num/den
-- the following doesn't compile

-- R has coerce: % -> Fraction Integer =>
-- q : Fraction Integer := coerce(r)@Fraction(Integer)
-- den : I := approxNthRoot(globalDigits * denom q ,n)$IntegerRoots(I)
-- num : I := approxNthRoot(globalDigits * numer q ,n)$IntegerRoots(I)
-- num/den

r -- this is nonsense, perhaps a Newton iteration for x**n-r here

)fin

-- for later use:

graeffe2 p ==
   -- substitute x by -x :
   eq : Equation UP := equation(monomial(1,1), monomial(-1$C,1))
   pp : UP := p*eval(p,eq)
   gp : UP := 0$UP
   while pp ^= 0 repeat
      i:NNI := (degree pp) quo (2::NNI)
      coef:C:=
      even? i => leadingCoefficient pp
      - leadingCoefficient pp
      gp := gp + monomial(coef,i)
      pp := reductum pp
   gp

shift2(p,c) ==
   d := degree p
   cc : C := 1
   coef := List C := [cc := c * cc for i in 1..d]
   coef := cons(1,coef)
   coef := [coefficient(p,i)*coef.(i+1) for i in 0..d]
   res : UP := 0
   for j in 0..d repeat
      cc := 0
      for i in j..d repeat
         cc := cc + coef.i * (binomial(i,j)$ICF :: R)
      res := res + monomial(cc,j)$UP
   res

scale2(p,c) ==
   d := degree p
   cc : C := 1
   coef := List C := [cc := c * cc for i in 1..d]
   coef := cons(1,coef)
   coef := [coefficient(p,i)*coef.(i+1) for i in 0..d]
   res : UP := 0
   for i in 0..d repeat res := res + monomial(coef.(i+1),i)$UP
   res

scale2: (UP,C) -> UP
shift2: (UP,C) -> UP
graeffe2 : UP -> UP
++ graeffe2 p determines q such that \(q(-z**2) = p(z)*p(-z)\).
++ Note that the roots of $q$ are the squares of the roots of $p$.

---

CRFP.dotabb

"CRFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CRFP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CRFP" -> "COMPCAT"

---

package CMPLXRT ComplexRootPackage

--- ComplexRootPackage.input ---

)set break resume
)sys rm -f ComplexRootPackage.output
)spool ComplexRootPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexRootPackage
--E 1

)spool
)lisp (bye)

---

--- ComplexRootPackage.help ---

====================================================================
ComplexRootPackage examples
====================================================================

This package provides functions complexZeros for finding the complex zeros of univariate polynomials with complex rational number coefficients. The results are to any user specified precision and are returned as either complex rational number or complex floating point numbers depending on the type of the second argument which specifies
the precision.

See Also:
- show ComplexRootPackage

---

ComplexRootPackage (CMPLXRT)

Exports:
- complexZeros

--- package CMPLXRT ComplexRootPackage ---

)abbrev package CMPLXRT ComplexRootPackage
++ Author: P. Gianni
++ Description:
++ This package provides functions complexZeros
++ for finding the complex zeros
++ of univariate polynomials with complex rational number coefficients.
++ The results are to any user specified precision and are returned
++ as either complex rational number or complex floating point numbers
++ depending on the type of the second argument which specifies the
++ precision.

ComplexRootPackage(UP,Par) : T == C where
  RN  ==> Fraction Integer
  I    ==> Integer
  NF   ==> Float
  UP   : UnivariatePolynomialCategory Complex Integer
  Par  : Join(Field, OrderedRing) -- will be Float or RN
  CP   ==> Complex Par
PCI ==> Polynomial Complex Integer

T == with
  complexZeros:(UP,Par) -> List CP
  ++ complexZeros(poly, eps) finds the complex zeros of the
  ++ univariate polynomial poly to precision eps with
  ++ solutions returned as complex floats or rationals
  ++ depending on the type of eps.

C == add
  complexZeros(p:UP,eps:Par):List CP ==
  x1:Symbol():=new()
  x2:Symbol():=new()
  vv:Symbol():=new()
  lpf:=factors factor(p)$ComplexFactorization(I,UP)
  ris:List CP:=empty()
  for pf in lpf repeat
    pp:=pf.factor pretend SparseUnivariatePolynomial Complex Integer
    q:PCI :=multivariate(pp,vv)
    q:=eval(q,vv,x1::PCI+complex(0,1)*(x2::PCI))
    p1:=map(real,q)$PolynomialFunctions2(Complex I,I)
    p2:=map(imag,q)$PolynomialFunctions2(Complex I,I)
    lz:=innerSolve([p1,p2],[],[x1,x2],
    eps)$InnerNumericFloatSolvePackage(I,Par,Par)
    ris:=append([complex(first z,second z) for z in lz],ris)
  ris

package CTRIGMNP ComplexTrigonometricManipulations

— ComplexTrigonometricManipulations.input —

)set break resume
CHAPTER 4. CHAPTER C

```lisp
(sys rm -f ComplexTrigonometricManipulations.output
spool ComplexTrigonometricManipulations.output
)set message test on
)set message auto off
)clear all

--) S 1 of 1
)show ComplexTrigonometricManipulations
--) E 1

)spool
)lisp (bye)

---

- ComplexTrigonometricManipulations.help ---

====================================================================
ComplexTrigonometricManipulations examples
====================================================================

ComplexTrigonometricManipulations provides function that compute the real and imaginary parts of complex functions.

See Also:
  o )show ComplexTrigonometricManipulations

---

ComplexTrigonometricManipulations (CTRIGMNP)

Exports:
  complexElementary complexForm complexNormalize imag
  real trigs
```
— package CTRIGMNP ComplexTrigonometricManipulations —

)abbrev package CTRIGMNP ComplexTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 11 June 1993
++ Date Last Updated: 14 June 1993
++ Description:
++ \spadtype{ComplexTrigonometricManipulations} provides function that
++ compute the real and imaginary parts of complex functions.

ComplexTrigonometricManipulations(R, F): Exports == Implementation where
R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
FunctionSpace Complex R)

SY ==> Symbol
FR ==> Expression R
K ==> Kernel F

Exports ==> with
complexNormalize: F -> F
++ complexNormalize(f) rewrites \spad{f} using the least possible number
++ of complex independent kernels.

complexNormalize: (F, SY) -> F
++ complexNormalize(f, x) rewrites \spad{f} using the least possible
++ number of complex independent kernels involving \spad{x}.

complexElementary: F -> F
++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
++ complex transcendental elementary functions: \spad{log, exp}.

complexElementary: (F, SY) -> F
++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
++ \spad{x} in terms of the 2 fundamental complex
++ transcendental elementary functions: \spad{log, exp}.
real : F -> FR
++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
++ function.
imag : F -> FR
++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
++ is a complex function.
real? : F -> Boolean
++ real?(f) returns \spad{true} if \spad{f = real f}.
trigs : F -> F
++ trigs(f) rewrites all the complex logs and exponentials
++ appearing in \spad{f} in terms of trigonometric functions.
complexForm: F -> Complex FR
++ complexForm(f) returns \spad{[real f, imag f]}. 
Implementation ==> add
import InnerTrigonometricManipulations(R, FR, F)
import ElementaryFunctionStructurePackage(Complex R, F)

rreal?: Complex R -> Boolean
kreal?: Kernel F -> Boolean
localexplogs : (F, F, List SY) -> F
real f == real complexForm f
imag f == imag complexForm f
rreal? r == zero? imag r
kreal? k == every?(real?, argument k)$List(F)
complexForm f == explogs2trigs f

trigs f ==
GF2FG explogs2trigs f

real? f ==
every?(rreal?, coefficients numer f)
and every?(rreal?, coefficients denom f) and every?(kreal?, kernels f)

localexplogs(f, g, lx) ==
trigs2explogs(g, [k for k in tower f
| is?(k, "tan"::SY) or is?(k, "cot"::SY)], lx)

complexElementary f ==
any?(x +-> has?(x, "rtrig"),
operators(g := realElementary f)$List(BasicOperator) =>
localexplogs(f, g, variables g))
g

complexElementary(f, x) ==
any?(y +-> has?(operator y, "rtrig"),
[k for k in tower(g := realElementary(f, x))
| member?(x, variables(k::F))]$List(K))$List(K) =>
localexplogs(f, g, [x])
g

complexNormalize(f, x) ==
any?(y +-> has?(operator y, "rtrig"),
[k for k in tower(g := realElementary(f, x))
| member?(x, variables(k::F))]$List(K))$List(K) =>
(rischNormalize(localexplogs(f, g, [x]), x).func)
rischNormalize(g, x).func

complexNormalize f ==
l := variables(g := realElementary f)
any?(y +-> has?(y, "rtrig"), operators g)$List(BasicOperator) =>
h := localexplogs(f, g, l)
for x in l repeat h := rischNormalize(h, x).func
h
for x in l repeat g := rischNormalize(g, x).func

g

---

<table>
<thead>
<tr>
<th>CTRIGMNP.dotabb</th>
</tr>
</thead>
</table>
"CTRIGMNP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CTRIGMNP"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"CDMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=CDMPCAT"]
"CTRIGMNP" -> "ACF"
"CTRIGMNP" -> "FS"
"CTRIGMNP" -> "CDMPCAT"

---

package ODECONST ConstantLODE

--- ConstantLODE.input ---

)set break resume
)sys rm -f ConstantLODE.output
)spool ConstantLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ConstantLODE
--E 1

)spool
)lisp (bye)

---

--- ConstantLODE.help ---

====================================================================
ConstantLODE examples
====================================================================
Solution of linear ordinary differential equations, constant coefficient case.

See Also:
- )show ConstantLODE

---

ConstantLODE (ODECONST)

Exports:
constDsolve

—— package ODECONST ConstantLODE ——

)abbrev package ODECONST ConstantLODE
++ Author: Manuel Bronstein
++ Date Created: 18 March 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ Solution of linear ordinary differential equations,
++ constant coefficient case.

ConstantLODE(R, F, L): Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
      LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R,
      TranscendentalFunctionCategory, PrimitiveFunctionCategory)
L: LinearOrdinaryDifferentialOperatorCategory F

Z ==> Integer
SY ==> Symbol
K ==> Kernel F
V ==> Vector F
M ==> Matrix F
SUP ==> SparseUnivariatePolynomial F

Exports ==> with
constDsolve: (L, F, SY) -> Record(particular:F, basis:List F)
++ constDsolve(op, g, x) returns \spad{[f, [y1,...,ym]]}
++ where f is a particular solution of the equation \spad{op y = g},
++ and the \spad{yi}'s form a basis for the solutions of \spad{op y = 0}.

Implementation ==> add
import ODETools(F, L)
import ODEIntegration(R, F)
import ElementaryFunctionSign(R, F)
import AlgebraicManipulations(R, F)
import FunctionSpaceIntegration(R, F)
import FunctionSpaceUnivariatePolynomialFactor(R, F, SUP)

homoBasis: (L, F) -> List F
quadSol : (SUP, F) -> List F
basisSqfr: (SUP, F) -> List F
basisSol : (SUP, Z, F) -> List F

constDsolve(op, g, x) ==
  b := homoBasis(op, x::F)
  [particularSolution(op, g, b, (f1:F):F +-> int(f1, x))::F, b]

homoBasis(op, x) ==
  p:SUP := 0
  while op ^= 0 repeat
    p := p + monomial(leadingCoefficient op, degree op)
    op := reductum op
  b:List(F) := empty()
  for ff in factors ffactor p repeat
    b := concat_!(b, basisSol(ff.factor, dec(ff.exponent), x))
b

basisSol(p, n, x) ==
  l := basisSqfr(p, x)
  zero? n => l
  ll := copy l
  xn := x::F
  for i in 1..n repeat
    l := concat_!(l, [xn * f for f in ll])
    xn := x * xn
  l

basisSqfr(p, x) ==
  -- one?(d := degree p) =>
  ((d := degree p) = 1) =>
\[
\exp(- \coefficient(p, 0) \times \text{leadingCoefficient } p) \]
d = 2 => quadSol(p, x)
[\exp(a \times x) \text{ for } a \text{ in } \text{rootsOf } p]

quadSol(p, x) ==
(u := \text{sign}(\delta := (b := \coefficient(p, 1))^2 - 4 \times
(a := \text{leadingCoefficient } p) \times (c := \coefficient(p, 0))))
\text{case } Z \text{ and } \text{negative?(u::Z)} =>
y := x / (2 \times a)
x := - b \times y
i := \text{rootSimp}(\sqrt{-\delta}) \times y
[\exp(r) \times \cos(i), \exp(r) \times \sin(i)]
[\exp(a \times x) \text{ for } a \text{ in } \text{zerosOf } p]

— ODECONST.dotabb —
"ODECONST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODECONST"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODECONST" -> "ACFS"

package COORDSYS CoordinateSystems

— CoordinateSystems.input —

)set break resume
)sys rm -f CoordinateSystems.output
)spool CoordinateSystems.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show CoordinateSystems
--E 1

)spool
)lisp (bye)

—
CoordinateSystems help

CoordinateSystems provides coordinate transformation functions for plotting. Functions in this package return conversion functions which take points expressed in other coordinate systems and return points with the corresponding Cartesian coordinates.

See Also:
o )show CoordinateSystems

Exports:
bipolar bipolarCylindrical cartesian conical cylindrical elliptic ellipticCylindrical oblateSpheroidal parabolic parabolicCylindrical paraboloidal polar prolateSpheroidal spherical toroidal

package COORDSYS CoordinateSystems

)abbrev package COORDSYS CoordinateSystems
++ Author: Jim Wen
++ Date Created: 12 March 1990
++ Date Last Updated: 19 June 1990, Clifton J. Williamson
++ Description:
++ CoordinateSystems provides coordinate transformation functions
++ for plotting. Functions in this package return conversion functions
++ which take points expressed in other coordinate systems and return points
++ with the corresponding Cartesian coordinates.

CoordinateSystems(R): Exports == Implementation where

R : Join(Field,TranscendentalFunctionCategory,RadicalCategory)
Pt => Point R
Exports => with
cartesian : Pt -> Pt
++ cartesian(pt) returns the Cartesian coordinates of point pt.
polar: Pt -> Pt
++ polar(pt) transforms pt from polar coordinates to Cartesian
++ coordinates: the function produced will map the point \(r, \theta\)
++ to \((r * \cos(\theta)), (r * \sin(\theta))\).
cylindrical: Pt -> Pt
++ cylindrical(pt) transforms pt from polar coordinates to Cartesian
++ coordinates: the function produced will map the point
++ \((r, \theta, z)\)
++ to \((r * \cos(\theta)), (r * \sin(\theta)), z\).
spherical: Pt -> Pt
++ spherical(pt) transforms pt from spherical coordinates to Cartesian
++ coordinates: the function produced will map the point
++ \((r, \theta, \phi)\)
++ to \((r * \sin(\phi) * \cos(\theta)), (r * \sin(\phi) * \sin(\theta)), r * \cos(\phi)\).
parabolic: Pt -> Pt
++ parabolic(pt) transforms pt from parabolic coordinates to Cartesian
++ coordinates: the function produced will map the point \((u, v)\)
++ to \((1/2 * (u^2 - v^2)), (u * v)\).
parabolicCylindrical: Pt -> Pt
++ parabolicCylindrical(pt) transforms pt from parabolic cylindrical
++ coordinates to Cartesian coordinates: the function produced will
++ map the point \((u, v, z)\) to \((1/2 * (u^2 - v^2)), (u * v), z\).
paraboloidal: Pt -> Pt
++ paraboloidal(pt) transforms pt from paraboloidal coordinates to
++ Cartesian coordinates: the function produced will map the point
++ \((u, v, \phi)\) to \((u * \sinh(\phi)), (u * \sin(\phi)), (1/2 * (u^2 - v^2))\).
elliptic: R -> (Pt -> Pt)
++ elliptic(a) transforms from elliptic coordinates to Cartesian
++ coordinates: \(a\) is a function which will map the
++ point \((u, v)\) to \((a * \cosh(u) * \cos(v)), (a * \sinh(u) * \sin(v))\).
ellipticCylindrical: R -> (Pt -> Pt)
++ ellipticCylindrical(a) transforms from elliptic cylindrical coordinates
++ to Cartesian coordinates: \(a\) is a function
++ which will map the point \((u, v, z)\) to \((a * \cosh(u) * \cos(v)), (a * \sinh(u) * \sin(v)), (z)\).
prolateSpheroidal: R -> (Pt -> Pt)
++ prolateSpheroidal(a) transforms from prolate spheroidal coordinates to
++ Cartesian coordinates: \(a\) is a function
++ which will map the point \((u, v, \phi)\) to \((a * \cosh(u) * \cos(v)), (a * \sinh(u) * \sin(v)), (1/2 * (u^2 - v^2))\).
++ Cartesian coordinates: \spad{prolateSpheroidal(a)} is a function
++ which will map the point \spad{(xi,eta,phi)} to
++ \spad{x = a*sinh(xi)*sin(eta)*cos(phi)}, \spad{y = a*sinh(xi)*sin(eta)*sin(phi)},
++ \spad{z = a*cosh(xi)*cos(eta)}.

oblateSpheroidal: R -> (Pt -> Pt)
++ oblateSpheroidal(a) transforms from oblate spheroidal coordinates to
++ Cartesian coordinates: \spad{oblateSpheroidal(a)} is a function which
++ will map the point \spad{(xi,eta,phi)} to \spad{x = a*sinh(xi)*sin(eta)*cos(phi)},
++ \spad{y = a*sinh(xi)*sin(eta)*sin(phi)}, \spad{z = a*cosh(xi)*cos(eta)}.

bipolar: R -> (Pt -> Pt)
++ bipolar(a) transforms from bipolar coordinates to Cartesian coordinates:
++ \spad{bipolar(a)} is a function which will map the point \spad{(u,v)} to
++ \spad{x = a*sinh(v)/(cosh(v)-cos(u))}, \spad{y = a*sin(u)/(cosh(v)-cos(u))}.

bipolarCylindrical: R -> (Pt -> Pt)
++ bipolarCylindrical(a) transforms from bipolar cylindrical coordinates
++ to Cartesian coordinates: \spad{bipolarCylindrical(a)} is a function which
++ will map the point \spad{(u,v,z)} to \spad{x = a*sinh(v)/(cosh(v)-cos(u))},
++ \spad{y = a*sin(u)/(cosh(v)-cos(u))}, \spad{z}.

toroidal: R -> (Pt -> Pt)
++ toroidal(a) transforms from toroidal coordinates to Cartesian
++ coordinates: \spad{toroidal(a)} is a function which will map the point
++ \spad{(u,v,phi)} to \spad{x = a*sinh(v)*cos(phi)/(cosh(v)-cos(u))},
++ \spad{y = a*sinh(v)*sin(phi)/(cosh(v)-cos(u))}, \spad{z = a*sin(u)/(cosh(v)-cos(u))}.

conical: (R,R) -> (Pt -> Pt)
++ conical(a,b) transforms from conical coordinates to Cartesian coordinates:
++ \spad{conical(a,b)} is a function which will map the point \spad{(lambda,mu,nu)} to
++ \spad{x = lambda*mu*nu/(a*b)},
++ \spad{y = lambda/a*sqrt((mu**2-a**2)*(nu**2-a**2)/(a**2-b**2))},
++ \spad{z = lambda/b*sqrt((mu**2-b**2)*(nu**2-b**2)/(b**2-a**2))}.

Implementation ==> add

cartesian pt ==
-- we just want to interpret the cartesian coordinates
-- from the first N elements of the point - so the
-- identity function will do
pt

polar pt0 ==
pt := copy pt0
r := elt(pt0,1); theta := elt(pt0,2)
pt.1 := r * cos(theta); pt.2 := r * sin(theta)
pt

cylindrical pt0 == polar pt0
-- apply polar transformation to first 2 coordinates

spherical pt0 ==
pt := copy pt0
r := elt(pt0,1); theta := elt(pt0,2); phi := elt(pt0,3)
\[ \text{pt.1} := r \times \sin(\phi) \times \cos(\theta); \text{pt.2} := r \times \sin(\phi) \times \sin(\theta) \]
\[ \text{pt.3} := r \times \cos(\phi) \]

\text{pt}

\text{parabolic pt0 ==}
\text{pt} := \text{copy pt0}
\text{u := elt(pt0,1); v := elt(pt0,2)}
\text{pt.1 := (u*u - v*v)/(2::R); pt.2 := u*v}
\text{pt}

\text{parabolicCylindrical pt0 == parabolic pt0}
-- apply parabolic transformation to first 2 coordinates

\text{paraboloidal pt0 ==}
\text{pt} := \text{copy pt0}
\text{u := elt(pt0,1); v := elt(pt0,2); phi := elt(pt0,3)}
\text{pt.1 := u*v*cos(\phi); pt.2 := u*v*sin(\phi); pt.3 := (u*u - v*v)/(2::R)}
\text{pt}

\text{elliptic a ==}
\text{x+->}
\text{pt} := \text{copy(x)}
\text{u := elt(x,1); v := elt(x,2)}
\text{pt.1 := a*cosh(u)*cos(v); pt.2 := a*sinh(u)*sin(v)}
\text{pt}

\text{ellipticCylindrical a == elliptic a}
-- apply elliptic transformation to first 2 coordinates

\text{prolateSpheroidal a ==}
\text{x+->}
\text{pt} := \text{copy(x)}
\text{xi := elt(x,1); eta := elt(x,2); phi := elt(x,3)}
\text{pt.1 := a*sinh(xi)*sin(eta)*cos(phi)}
\text{pt.2 := a*sinh(xi)*sin(eta)*sin(phi)}
\text{pt.3 := a*cosh(xi)*cos(eta)}
\text{pt}

\text{oblateSpheroidal a ==}
\text{x+->}
\text{pt} := \text{copy(x)}
\text{xi := elt(x,1); eta := elt(x,2); phi := elt(x,3)}
\text{pt.1 := a*sinh(xi)*sin(eta)*cos(phi)}
\text{pt.2 := a*cosh(xi)*cos(eta)*sin(phi)}
\text{pt.3 := a*sinh(xi)*sin(eta)}
\text{pt}

\text{bipolar a ==}
\text{x+->}
\text{pt} := \text{copy(x)}
\[ u := \text{elt}(x,1); \ v := \text{elt}(x,2) \]
\[ \text{pt.1} := a \sinh(v)/(\cosh(v)-\cos(u)) \]
\[ \text{pt.2} := a \sin(u)/(\cosh(v)-\cos(u)) \]
\[ \text{pt} \]

\text{bipolarCylindrical} a \Rightarrow \text{bipolar} a \\
\quad \text{-- apply bipolar transformation to first 2 coordinates} \\

\text{toroidal} a \Rightarrow \\
x+-> \\
\quad \text{pt} := \text{copy}(x) \\
\quad u := \text{elt}(x,1); \ v := \text{elt}(x,2); \ \phi := \text{elt}(x,3) \\
\quad \text{pt.1} := a \sinh(v)\cos(\phi)/(\cosh(v)-\cos(u)) \\
\quad \text{pt.2} := a \sinh(v)\sin(\phi)/(\cosh(v)-\cos(u)) \\
\quad \text{pt.3} := a \sin(u)/(\cosh(v)-\cos(u)) \\
\quad \text{pt} \\

\text{conical}(a,b) \Rightarrow \\
x+-> \\
\quad \text{pt} := \text{copy}(x) \\
\quad \lambda := \text{elt}(x,1); \ \mu := \text{elt}(x,2); \ \nu := \text{elt}(x,3) \\
\quad \text{pt.1} := \lambda \mu \nu/(a b) \\
\quad \text{pt.2} := \lambda/a \sqrt{(\mu^2-a^2)(\nu^2-a^2)/(a^2-b^2)} \\
\quad \text{pt.3} := \lambda/b \sqrt{(\mu^2-b^2)(\nu^2-b^2)/(b^2-a^2)} \\
\quad \text{pt} \\

<table>
<thead>
<tr>
<th>COORDSYS.dotabb</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;COORDSYS&quot; [color=&quot;#FF4488&quot;,href=&quot;bookvol10.4.pdf#nameddest=COORDSYS&quot;]</td>
</tr>
<tr>
<td>&quot;PTCAT&quot; [color=&quot;#4488FF&quot;,href=&quot;bookvol10.2.pdf#nameddest=PTCAT&quot;]</td>
</tr>
<tr>
<td>&quot;COORDSYS&quot; -&gt; &quot;PTCAT&quot;</td>
</tr>
</tbody>
</table>

---

package CRAPACK CRApackage

--- CRApackage.input ---

\)set break resume
\)sys rm -f CRApackage.output
\)spool CRApackage.output
\)set message test on
CRApkg is a package that provides functions for working with modular arithmetic. It exports several functions, including:

- modTree
- chineseRemainder
- multiEuclideanTree

Here is an example of how to use CRApkg:

```lisp
)read crapkg
```

### Example

```
> CRApkg
> CRApkg
> set message auto off
> clear all

--S 1 of 1
> show CRApkg
--E 1

> spool
> lisp (bye)
```

---

--- CRApkg.help ---

--- CRApkg examples ---

This package has no documentation

See Also:
- `)show CRApkg`
---

**CRApkg (CRAPACK)**

Exports:

- modTree
- chineseRemainder
- multiEuclideanTree

--- package CRAPACK CRApkg ---

`)abbrev package CRAPACK CRApkg`
++ Description:
++ This package has no documentation

CRAPackage(R:EuclideanDomain): Exports == Implementation where
Exports == with
modTree: (R,List R) -> List R
  ++ modTree(r,l) undocumented{}
chineseRemainder: (List R, List R) -> R
  ++ chineseRemainder(lv,lm) returns a value \axiom{v} such that, if
  ++ \axiom{x} is \axiom{lv.i} modulo \axiom{lm.i} for all \axiom{i}, then
  ++ \axiom{x} is \axiom{v} modulo \axiom{lm(1)*lm(2)*...*lm(n)}.
chineseRemainder: (List List R, List R) -> List R
  ++ chineseRemainder(llv,lm) returns a list of values, each of which
  ++ corresponds to the Chinese remainder of the associated element of
  ++ \axiom{llv} and \axiom{lm}. This is more efficient than applying
  ++ chineseRemainder several times.
multiEuclideanTree: (List R, R) -> List R
  ++ multiEuclideanTree(l,r) undocumented{}
Implementation == add

BB:=BalancedBinaryTree(R)
x:BB

-- Definition for modular reduction mapping with several moduli
modTree(a,lm) ==
t := balancedBinaryTree(#lm, 0$R)
setleaves_!(t,lm)
mapUp_!(t,"*"
leaves mapDown_!(t, a, "rem")

chineseRemainder(lv:List(R), lm:List(R)):R ==
  #lm ^= #lv => error "lists of moduli and values not of same length"
x := balancedBinaryTree(#lm, 0$R)
x := setleaves_!(x, lm)
mapUp_!(x,"*"
    y := balancedBinaryTree(#lm, 1$R)
    y := mapUp_!(copy y,x,(a,b,c,d)+->a*d + b*c)
    (u := extendedEuclidean(value y, value x,1)) case "failed" =>
      error "moduli not relatively prime"
    inv := u . coef1
    linv := modTree(inv, lm)
l := [(u*v) rem m for v in lv for u in linv for m in lm]
y := setleaves_!(y,1)
    value(mapUp_!(y, x, (a,b,c,d)+->a*d + b*c)) rem value(x)

chineseRemainder(llv:List List(R), lm:List(R))::List(R) ==
x := balancedBinaryTree(#lm, 0$R)
x := setleaves_!(x, lm)
mapUp_!(x,"*"
    y := balancedBinaryTree(#lm, 1$R)
```plaintext
y := mapUp_!(copy y, x, (a,b,c,d) +> a*d + b*c)
(u := extendedEuclidean(value y, value x, 1)) case "failed" =>
  error "moduli not relatively prime"
inv := u . coef1
linv := modTree(inv, lm)
retVal: List(R) := []
for lv in llv repeat
  l := [(u3*v) rem m for v in lv for u3 in linv for m in lm]
y := setleaves!(y, l)
retVal :=
  cons(value(mapUp!(y, x, (a,b,c,d) +> a*d+b*c)) rem value(x), retVal)
reverse retVal

extEuclidean: (R, R, R) -> List R
extEuclidean(a, b, c) ==
  u := extendedEuclidean(a, b, c)
  u case "failed" => error [c, " not spanned by ", a, " and ", b]
  [u.coef2, u.coef1]

multiEuclideanTree(fl, rhs) ==
  x := balancedBinaryTree(#fl, rhs)
  x := setleaves_!(x, fl)
  mapUp_!(x,"*")
  leaves mapDown_!(x, rhs, extEuclidean)
```

— CRAPACK.dotabb —

"CRAPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CRAPACK"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"CRAPACK" -> "FLAGG"

package CYCLES CycleIndicators

— CycleIndicators.input —

)set break resume
)sys rm -f CycleIndicators.output
)spool CycleIndicators.output
)set message test on
)set message auto off
```
)clear all
--S 1 of 47
complete 1
--R
--R
--R (1) (1)
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 1

--S 2 of 47
complete 2
--R
--R
--R 1 1 2
--R (2) - (2) + - (1 )
--R 2 2
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 2

--S 3 of 47
complete 3
--R
--R
--R 1 1 1 3
--R (3) - (3) + - (2 1) + - (1 )
--R 3 2 6
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 3

--S 4 of 47
complete 7
--R
--R
--R (4)
--R 1 1 1 1 1 2 1 1 1 1 3
--R - (7) + - (6 1) + -- (5 2) + -- (5 1 ) + -- (4 3) + - (4 2 1) + -- (4 1 )
--R 7 6 10 10 12 8 24
--R +
--R 1 2 1 2 1 2 1 4 1 3 1 2 3
--R -- (3 1) + -- (3 2 ) + -- (3 2 1 ) + -- (3 1 ) + -- (2 1) + -- (2 1 )
--R 18 24 12 72 48 48
--R +
--R 1 5 1 7
--R --- (2 1) + ---- (1 )
--R 240 5040
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 4

--S 5 of 47
elementary 7
\begin{verbatim}
--E 8
--S 9 of 47
graphs 5
--R
--R
--R (9)
--R  1  1  2  1  2  1  3  1  4  2  1  3  4  1  10
--R - (6 3 1) + - (5 ) + - (4 2) + - (3 1) + - (2 1) + -- (2 1) + --- (1 )
--R  6  5  4  6  8 12 120
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 9

--S 10 of 47
cap(complete 2**2, complete 2*complete 1**2)
--R
--R
--R (10) 4
--R Type: Fraction(Integer)
--E 10

--S 11 of 47
cap(elementary 2**2, complete 2*complete 1**2)
--R
--R
--R (11) 2
--R Type: Fraction(Integer)
--E 11

--S 12 of 47
cap(complete 3*complete 2*complete 1,complete 2**2*complete 1**2)
--R
--R
--R (12) 24
--R Type: Fraction(Integer)
--E 12

--S 13 of 47
cap(elementary 3*elementary 2*elementary 1,complete 2**2*complete 1**2)
--R
--R
--R (13) 8
--R Type: Fraction(Integer)
--E 13

--S 14 of 47
cap(complete 3*complete 2*complete 1,elementary 2**2*elementary 1**2)
--R
--R
--R (14) 8
\end{verbatim}
eval(cup(complete 3*complete 2*complete 1, _
   cup(complete 2**2*complete 1**2,complete 2**3)))

\[\text{(15) 1500}\]

\[\text{square:=dihedral 4}\]

\[\text{(16) - (4) + - (2) + - (2 1) + - (1) + 4 8 4 8}\]

\[\text{cap(complete 2**2,square)}\]

\[\text{(17) 2}\]

\[\text{cap(complete 3*complete 2**2,dihedral 7)}\]

\[\text{(18) 18}\]

\[\text{cap(graphs 5,complete 7*complete 3)}\]

\[\text{(19) 4}\]

\[s(x) == powerSum(x)\]

\[\text{Type: Void}\]
cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)

--R
--R Compiling function s with type PositiveInteger ->
--R SymmetricPolynomial(Fraction(Integer))
--R
--R (21) - (4 ) + - (3 1 ) + - (2 ) + -- (1 )
--R 4 3 8 24
--R Type: SymmetricPolynomial(Fraction(Integer))

--- E 21

cap(complete 4**2,cube)
--R
--R (22) 7
--R Type: Fraction(Integer)
--E 22

--- S 22 of 47

cap(complete 2**3*complete 1**2,wreath(elementary 4,elementary 2))
--R
--R (23) 7
--R Type: Fraction(Integer)
--E 23

--- S 23 of 47

cap(complete 2**3*complete 1**2,wreath(elementary 4,complete 2))
--R
--R (24) 17
--R Type: Fraction(Integer)
--E 24

--- S 24 of 47

cap(complete 2**3*complete 1**2,wreath(complete 4,elementary 2))
--R
--R (25) 10
--R Type: Fraction(Integer)
--E 25

--- S 25 of 47

cap(complete 2**3*complete 1**2,wreath(complete 4,complete 2))
--R
--R

--- S 26 of 47

--R
ZeroOrOne : INT -> ULS(FRAC INT, 'x, 0)

Integers : INT -> ULS(FRAC INT, 'x, 0)

ZeroOrOne n == 1+x**n

Integers n == 1/(1-x**n)

ZeroOrOne 5

Integers 5
--R Compiling function Integers with type Integer ->
--R UnivariateLaurentSeries(Fraction(Integer),x,0)
--R
--R  5  10  11
--R (33) 1 + x + x + O(x)
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 33

--S 34 of 47
)expose EVALCYC
--R
--I EvaluateCycleIndicators is now explicitly exposed in frame frame0
--E 34

--S 35 of 47
eval(ZeroOrOne, graphs 5)
--R
--R
--R  2  3  4  5  6  7  8  9  10  11
--R (34) 1 + x + 2x + 4x + 6x + 6x + 4x + 2x + x + x + O(x)
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 35

--S 36 of 47
eval(ZeroOrOne, dihedral 8)
--R
--R
--R  2  3  4  5  6  7  8
--R (35) 1 + x + 4x + 5x + 8x + 5x + 4x + x + x
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 36

--S 37 of 47
eval(Integers, complete 4)
--R
--R
--R (36)
--R  2  3  4  5  6  7  8  9  10  11
--R 1 + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x + O(x)
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 37

--S 38 of 47
eval(Integers, elementary 4)
--R
--R
--R (37)
--R  6  7  8  9  10  11  12  13  14  15  16
--R x + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x
--R +
eval(ZeroOrOne,cube)

\[
\begin{align*}
\text{(38))} & \quad 1 + x + 3x + 3x + 7x + 3x + 3x + x + x \\
\text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)}
\end{align*}
\]

eval(Integers,cube)

\[
\begin{align*}
\text{(39))} & \quad 1 + x + 4x + 7x + 21x + 37x + 85x + 151x + 292x + 490x + 848x \\
\text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)}
\end{align*}
\]

eval(Integers,graphs 5)

\[
\begin{align*}
\text{(40))} & \quad 1 + x + 3x + 7x + 17x + 35x + 76x + 149x + 291x + 539x + 974x \\
\text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)}
\end{align*}
\]

eval(ZeroOrOne,graphs 15)

\[
\begin{align*}
\text{(41))} & \quad 1 + x + 2x + 5x + 11x + 26x + 68x + 177x + 496x + 1471x + 4583x \\
\text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)}
\end{align*}
\]
package cycles cycleindicators

--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 42

--S 43 of 47

cap(dihedral 30, complete 7*complete 8*complete 5*complete 10)
--R
--R
--R (42) 4995897238320
--R Type: Fraction(Integer)
--E 43

--S 44 of 47
sf3221:= SFunction [3,2,2,1]
--R
--R
--R (43)
--R 1 1 2 1 2 1 1 4 1 2
--R -- (6 2) -- (6 1) -- (4) + -- (4 3 1) + -- (4 1) -- (3 2)
--R 12 12 16 12 24 36
--R +
--R 1 2 2 1 2 1 3 1 5 1 4 1 3 2
--R -- (3 1) -- (3 2 1) -- (3 2 1) -- (3 1) -- (2) + -- (2 1)
--R 36 24 36 72 192 48
--R +
--R 1 2 4 1 6 1 8
--R -- (2 1) -- (2 1) + -- (1)
--R 96 144 576
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 44

--S 45 of 47
cap(sf3221, complete 2**4)
--R
--R
--R (44) 3
--R Type: Fraction(Integer)
--E 45

--S 46 of 47
cap(sf3221, powerSum 1**8)
--R
--R
--R (45) 70
--R Type: Fraction(Integer)
--E 46

--S 47 of 47
eval(Integers, sf3221)
--R
--R
CHAPTER 4. CHAPTER C

---R (46)
---R 9 10 11 12 13 14 15 16 17 18
---R x + 3x + 7x + 14x + 27x + 47x + 79x + 126x + 196x + 294x
---R +
---R 19 20
---R 432x + O(x )
---E 47
}spool
{lisp (bye)

--- CycleIndicators.help ---

====================================================================
CycleIndicators examples
====================================================================

Polya-Redfield enumeration by cycle indices.

This section is based upon the paper J. H. Redfield, ‘‘The Theory of
Group-Reduced Distributions’’, American J. Math.,49 (1927) 433-455,
and is an application of group theory to enumeration problems. It is
a development of the work by P. A. MacMahon on the application of
symmetric functions and Hammond operators to combinatorial theory.

The theory is based upon the power sum symmetric functions s(i) which
are the sum of the i-th powers of the variables. The cycle index of a
permutation is an expression that specifies the sizes of the cycles of
a permutation, and may be represented as a partition. A partition of
a non-negative integer n is a collection of positive integers called
its parts whose sum is n. For example, the partition (3^2 2 1^2) will
be used to represent s^2_3 s_2 s^2_1 and will indicate that the
permutation has two cycles of length 3, one of length 2 and two of
length 1. The cycle index of a permutation group is the sum of the
cycle indices of its permutations divided by the number of
permutations. The cycle indices of certain groups are provided.

The operation complete returns the cycle index of the symmetric group
of order n for argument n. Alternatively, it is the n-th complete
homogeneous symmetric function expressed in terms of power sum
symmetric functions.

complete 1
(1)

Type: SymmetricPolynomial Fraction Integer

complete 2
The operation `elementary` computes the n-th elementary symmetric function for argument n.

\[ \text{elementary 7} \]
\[ \begin{array}{cccccccccccc}
1 & 1 & 2 & \\
- (2) + - (1) & \\
2 & 2 & \\
\end{array} \]

Type: SymmetricPolynomial Fraction Integer

The operation `alternating` returns the cycle index of the alternating group having an even number of even parts in each cycle partition.

\[ \text{alternating 7} \]
\[ \begin{array}{cccccccccccc}
2 & 1 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 4 & 1 & 2 & 3 \\
- (7) + - (5 1) + - (4 2 1) + - (3 1) + - (2 1) + - (2 1) & \\
7 & 5 & 4 & 9 & 12 & 36 & 24 & \\
\end{array} \]

Type: SymmetricPolynomial Fraction Integer
The operation cyclic returns the cycle index of the cyclic group.

\[
\begin{align*}
1 & \quad 7 \\
\text{cyclic 7} & \\
6 & \quad 1 \quad 7 \\
- (7) & \quad + \quad - (1) \\
7 & \quad 7 \\
\end{align*}
\]

Type: SymmetricPolynomial Fraction Integer

The operation dihedral is the cycle index of the dihedral group.

\[
\begin{align*}
1 & \quad 3 \quad 1 \quad 7 \\
dihedral 7 & \\
3 & \quad 1 \quad 3 \quad 1 \quad 7 \\
- (7) & \quad + \quad - (2 \ 1) \quad + \quad - (1) \\
7 & \quad 2 \quad 14 \\
\end{align*}
\]

Type: SymmetricPolynomial Fraction Integer

The operation graphs for argument n returns the cycle index of the group of permutations on the edges of the complete graph with n nodes induced by applying the symmetric group to the nodes.

\[
\begin{align*}
1 & \quad 2 \quad 1 \quad 2 \quad 1 \quad 3 \quad 1 \quad 4 \quad 2 \quad 1 \quad 3 \quad 4 \quad 1 \quad 10 \\
graphs 5 & \\
6 & \quad 5 \quad 4 \quad 6 \quad 8 \quad 12 \quad 120 \\
- (6 \ 3 \ 1) & \quad + \quad - (5) \quad + \quad - (4 \ 2) \quad + \quad - (3 \ 1) \quad + \quad - (2 \ 1) \quad + \quad - (2 \ 1) \quad + \quad - (1) \\
\end{align*}
\]

Type: SymmetricPolynomial Fraction Integer

The cycle index of a direct product of two groups is the product of the cycle indices of the groups. Redfield provided two operations on two cycle indices which will be called "cup" and "cap" here. The cup of two cycle indices is a kind of scalar product that combines monomials for permutations with the same cycles. The cap operation provides the sum of the coefficients of the result of the cup operation which will be an integer that enumerates what Redfield called group-reduced distributions.

We can, for example, represent complete 2 * complete 2 as the set of objects a a b b and complete 2 * complete 1 * complete 1 as c c d e.

This integer is the number of different sets of four pairs.

\[
\begin{align*}
\text{cap(complete 2**2, complete 2*complete 1**2)} & \\
4 & \\
\text{Type: Fraction Integer} \\
\end{align*}
\]

For example,
This integer is the number of different sets of four pairs no two pairs being equal.

\[
\text{cap}(\text{elementary } 2^{*2}, \text{ complete } 2*\text{complete } 1^{*2})
\]
\[
2
\]
Type: Fraction Integer

For example,

\[
\begin{align*}
a & a \ b \ b \\
c & d \ e \ c \\
\end{align*}
\]

In this case the configurations enumerated are easily constructed, however the theory merely enumerates them providing little help in actually constructing them.

Here are the number of 6-pairs, first from a a b b c, second from d d e e f g.

\[
\text{cap}(\text{complete } 3*\text{complete } 2*\text{complete } 1, \text{complete } 2^{*2}*\text{complete } 1^{*2})
\]
\[
24
\]
Type: Fraction Integer

Here it is again, but with no equal pairs.

\[
\text{cap}(\text{elementary } 3*\text{elementary } 2*\text{elementary } 1, \text{complete } 2^{*2}*\text{complete } 1^{*2})
\]
\[
8
\]
Type: Fraction Integer

\[
\text{cap}(\text{complete } 3*\text{complete } 2*\text{complete } 1, \text{elementary } 2^{*2}*\text{elementary } 1^{*2})
\]
\[
8
\]
Type: Fraction Integer

The number of 6-triples, first from a a b b c, second from d d e e f g, third from h h i i j j.

\[
\text{eval}(\text{cup}(\text{complete } 3*\text{complete } 2*\text{complete } 1, \text{cup}(\text{complete } 2^{*2}*\text{complete } 1^{*2}, \text{complete } 2^{*3})))
\]
\[
1500
\]
Type: Fraction Integer

The cycle index of vertices of a square is dihedral 4.

\[
square:=\text{dihedral} \ 4
\]
\[
1 \ 3 \ 2 \ 1 \ 2 \ 1 \ 4
\]
\[
- (4) + - (2 ) + - (2 1 ) + - (1 )
\]
\[
4 \ 8 \ 4 \ 8
\]
Type: Symmetric Polynomial Fraction Integer
The number of different squares with 2 red vertices and 2 blue vertices.
\[
\text{cap(complete } 2**2, \text{square)}
\]
\[2\]
Type: Fraction Integer

The number of necklaces with 3 red beads, 2 blue beads and 2 green beads.
\[
\text{cap(complete } 3*\text{complete } 2**2, \text{dihedral } 7)
\]
\[18\]
Type: Fraction Integer

The number of graphs with 5 nodes and 7 edges.
\[
\text{cap(graphs } 5, \text{complete } 7*\text{complete } 3)
\]
\[4\]
Type: Fraction Integer

The cycle index of rotations of vertices of a cube.
\[
s(x) \equiv \text{powerSum}(x)
\]
Type: Void
\[
cube:=((1/24)*(s 1**8+9*s 2**4+8*s 3**2*s 1**2+6*s 4**2)
1 2 1 2 2 3 4 1 8
- (4 ) + - (3 1 ) + - (2 ) + -- (1 )
4 3 8 24
Type: SymmetricPolynomial Fraction Integer
\]

The number of cubes with 4 red vertices and 4 blue vertices.
\[
\text{cap(complete } 4**2, \text{cube)}
\]
\[7\]
Type: Fraction Integer

The number of labeled graphs with degree sequence 2 2 2 1 1 with no loops or multiple edges.
\[
\text{cap(complete } 2**3*\text{complete } 1**2, \text{wreath(elementary } 4, \text{elementary } 2))
\]
\[7\]
Type: Fraction Integer

Again, but with loops allowed but not multiple edges.
\[
\text{cap(complete } 2**3*\text{complete } 1**2, \text{wreath(elementary } 4, \text{complete } 2))
\]
\[17\]
Type: Fraction Integer

Again, but with multiple edges allowed, but not loops
Again, but with both multiple edges and loops allowed
\[\text{cap}(\text{complete } 2^*3 \text{complete } 1^*2, \text{wreath}(\text{complete } 4, \text{elementary } 2))\]
\[23\]
\[\text{Type: Fraction Integer}\]

Having constructed a cycle index for a configuration we are at liberty to evaluate the \(s_i\) components any way we please. For example we can produce enumerating generating functions. This is done by providing a function \(f\) on an integer \(i\) to the value required of \(s_i\), and then evaluating \(\text{eval}(f, \text{cycleindex})\).

\[
x: \text{ULS}(\text{FRAC INT, 'x, 0}) := 'x
\]

\[\text{x}\]
\[\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}\]

\[
\text{ZeroOrOne: INT -> ULS(\text{FRAC INT, 'x, 0})}
\]

\[\text{Type: Void}\]

\[
\text{Integers: INT -> ULS(\text{FRAC INT, 'x, 0})}
\]

\[\text{Type: Void}\]

For the integers 0 and 1, or two colors.

\[
\text{ZeroOrOne n == 1+x**n}
\]

\[\text{Type: Void}\]

\[
\text{ZeroOrOne 5}
\]

\[5\]

\[1 + x\]

\[\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}\]

For the integers 0, 1, 2, ... we have this.

\[
\text{Integers n == 1/(1-x**n)}
\]

\[\text{Type: Void}\]

\[
\text{Integers 5}
\]

\[5\]

\[10\]

\[11\]

\[1 + x + x + O(x)\]

\[\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}\]

The coefficient of \(x^n\) is the number of graphs with 5 nodes and \(n\) edges.

Note that there is an \text{eval} function that takes two arguments. It has the
signature:

\[(\text{Integer} \rightarrow \text{D1}), \text{SymmetricPolynomial Fraction Integer} \rightarrow \text{D1}\]

from EvaluateCycleIndicators D1 if D1 has ALGEBRA FRAC INT

This function is not normally exposed (it will not normally be considered in the list of eval functions) as it is only useful for this particular domain. To use it we ask that it be considered thus:

>expose EVALCYC

and now we can use it:

\[
eval(\text{ZeroOrOne}, \text{graphs 5})
\]

\[
1 + x + 2x + 4x + 6x + 6x + 4x + 2x + x + x + O(x^8)
\]

Type: UnivariateLaurentSeries(Fraction Integer, x, 0)

The coefficient of \(x^n\) is the number of necklaces with \(n\) red beads and \(n-8\) green beads.

\[
eval(\text{ZeroOrOne}, \text{dihedral 8})
\]

\[
1 + x + 4x + 5x + 8x + 5x + 4x + x + x
\]

Type: UnivariateLaurentSeries(Fraction Integer, x, 0)

The coefficient of \(x^n\) is the number of partitions of \(n\) into 4 or fewer parts.

\[
eval(\text{Integers}, \text{complete 4})
\]

\[
1 + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x + O(x^{11})
\]

Type: UnivariateLaurentSeries(Fraction Integer, x, 0)

The coefficient of \(x^n\) is the number of partitions of \(n\) into 4 boxes containing ordered distinct parts.

\[
eval(\text{Integers}, \text{elementary 4})
\]

\[
x + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x + 17
\]

Type: UnivariateLaurentSeries(Fraction Integer, x, 0)

The coefficient of \(x^n\) is the number of different cubes with \(n\) red vertices and \(8-n\) green ones.

\[
eval(\text{ZeroOrOne}, \text{cube})
\]

\[
1 + x + 3x + 3x + 7x + 3x + 3x + x + x
\]
The coefficient of $x^n$ is the number of different cubes with integers on the vertices whose sum is $n$.

\begin{verbatim}
  eval(Integers,cube)
  2 3 4 5 6 7 8 9 10
  1 + x + 4x + 7x + 21x + 37x + 85x + 151x + 292x + 490x + 848x
  +
  11
  O(x^11)
\end{verbatim}

The coefficient of $x^n$ is the number of graphs with 5 nodes and with integers on the edges whose sum is $n$. In other words, the enumeration is of multigraphs with 5 nodes and $n$ edges.

\begin{verbatim}
  eval(Integers,graphs 5)
  2 3 4 5 6 7 8 9 10
  1 + x + 3x + 7x + 17x + 35x + 76x + 149x + 291x + 539x + 974x
  +
  11
  O(x^11)
\end{verbatim}

Graphs with 15 nodes enumerated with respect to number of edges.

\begin{verbatim}
  eval(ZeroOrOne,graphs 15)
  2 3 4 5 6 7 8 9 10
  1 + x + 2x + 5x + 11x + 26x + 68x + 177x + 496x + 1471x + 4583x
  +
  11
  O(x^11)
\end{verbatim}

Necklaces with 7 green beads, 8 white beads, 5 yellow beads and 10 red beads.

\begin{verbatim}
  cap(dihedral 30,complete 7*complete 8*complete 5*complete 10)
  4995897238320
\end{verbatim}

The operation SFunction is the S-function or Schur function of a partition written as a descending list of integers expressed in terms of power sum symmetric functions.

In this case the argument partition represents a tableau shape. For example $3,2,2,1$ represents a tableau with three boxes in the first row, two boxes in the second and third rows, and one box in the fourth row. SFunction $[3,2,2,1]$ counts the number of different tableaux of
shape 3, 2, 2, 1 filled with objects with an ascending order in the columns and a non-descending order in the rows.

\[ sf3221 := \text{SFUnction}[3,2,2,1] \]

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 1 & 2 & 1 & 1 & 4 & 1 & 2 \\
\text{-- (6 2) -- (6 1) -- (4) + -- (4 3 1) + -- (4 1) -- (3 2)} \\
12 & 12 & 16 & 12 & 24 & 36 \\
+ \\
1 & 2 & 2 & 1 & 2 & 1 & 3 & 1 & 5 & 1 & 4 & 1 & 3 & 2 \\
\text{-- (3 1) -- (3 2 1) -- (3 2 1) -- (3 1) -- (2) + -- (2 1)} \\
36 & 24 & 16 & 12 & 24 & 36 & 72 & 192 & 48 \\
+ \\
1 & 2 & 4 & 1 & 6 & 1 & 8 \\
\text{-- (2 1) -- (2 1) + (1)} \\
96 & 144 & 576
\end{array}
\]

Type: SymmetricPolynomial Fraction Integer

This is the number filled with a a b b c c d d.

\[ \text{cap(sf3221, complete 2**4)} \]

3

Type: Fraction Integer

The configurations enumerated above are:

a a b a a c a a d
b c b b b b
c d c d c c
d d d

This is the number of tableaux filled with 1..8.

\[ \text{cap(sf3221, powerSum 1**8)} \]

70

Type: Fraction Integer

The coefficient of \( x^n \) is the number of column strict reverse plane partitions of \( n \) of shape 3 2 2 1.

\[ \text{eval(Integers, sf3221)} \]

\[
\begin{array}{cccccccccccccccc}
9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
x & + & 3x & + & 7x & + & 14x & + & 27x & + & 47x & + & 79x & + & 126x & + & 196x \\
+ \\
18 & 19 & 20 \\
294x & + & 432x & + & 0(x) \\
\end{array}
\]

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The smallest is

0 0 0
CycleIndicators (CYCLES)

Exports:
alternating cap complete cup cyclic
dihedral elementary eval graphs powerSum
SFunction skewSFunction wreath

package CYCLES CycleIndicators
SPOL ==> SymmetricPolynomial
PTN ==> Partition
RN ==> Fraction Integer
FR ==> Factored Integer
h ==> complete
s ==> powerSum
--a ==> elementary
alt ==> alternating
cyc ==> cyclic
dih ==> dihedral
ev == eval
Exports ==> with

complete: I -> SPOL RN
++\(\text{complete } n\) is the \(\text{spad}(n)\) th complete homogeneous
++ symmetric function expressed in terms of power sums.
++ Alternatively it is the cycle index of the symmetric
++ group of degree \(n\).

powerSum: I -> SPOL RN
++\(\text{powerSum } n\) is the \(\text{spad}(n)\) th power sum symmetric
++ function.

elementary: I -> SPOL RN
++\(\text{elementary } n\) is the \(\text{spad}(n)\) th elementary symmetric
++ function expressed in terms of power sums.

-- s2h: I -> SPOL RN--s to h

alternating: I -> SPOL RN
++\(\text{alternating } n\) is the cycle index of the
++ alternating group of degree \(n\).

cyclic: I -> SPOL RN  --cyclic group
++\(\text{cyclic } n\) is the cycle index of the
++ cyclic group of degree \(n\).

dihedral: I -> SPOL RN  --dihedral group
++\(\text{dihedral } n\) is the cycle index of the
++ dihedral group of degree \(n\).

graphs: I -> SPOL RN
++\(\text{graphs } n\) is the cycle index of the group induced on
++ the edges of a graph by applying the symmetric function to the
++ \(n\) nodes.

cap: (SPOL RN,SPOL RN) -> RN
++\(\text{cap}(s1,s2)\), introduced by Redfield,
++ is the scalar product of two cycle indices.
cup: (SPOL RN, SPOL RN) -> SPOL RN
++ @spad{cup(s1,s2)}, introduced by Redfield,
++ is the scalar product of two cycle indices, in which the
++ power sums are retained to produce a cycle index.

eval: SPOL RN -> RN
++ @spad{eval s} is the sum of the coefficients of a cycle index.

wreath: (SPOL RN, SPOL RN) -> SPOL RN
++ @spad{wreath(s1,s2)} is the cycle index of the wreath product
++ of the two groups whose cycle indices are @spad{s1} and
++ @spad{s2}.

SFunction: L I -> SPOL RN
++ @spad{SFunction(li)} is the S-function of the partition @spad{li}
++ expressed in terms of power sum symmetric functions.

skewSFunction: (L I, L I) -> SPOL RN
++ @spad{skewSFunction(li1,li2)} is the S-function
++ of the partition difference @spad{li1 - li2}
++ expressed in terms of power sum symmetric functions.

Implementation ==> add
import PartitionsAndPermutations
import IntegerNumberTheoryFunctions

trm: PTN -> SPOL RN
trm pt == monomial(inv(pdct(pt) :: RN),pt)

list: Stream L I -> L L I
list st == entries complete st

complete i ==
  if i=0
  then 1
  else if i<0
    then 0
    else
      _+/[trm(partition pt) for pt in list(partitions i)]

even?: L I -> B
even? li == even?( #([i for i in li | even? i])

alt i ==
  2 * _+/[trm(partition li) for li in list(partitions i) | even? li]
elementary i ==
  if i=0
    then 1
    else if i<0
\[
\text{then 0} \\
\text{else} \\
\text{+}[(\text{spol := trm(partition pt); even? pt } \Rightarrow \text{ spol}; \ -\text{spol}) \\
\text{for pt in list(partitions i)}]
\]

divisors: \( I \rightarrow L I \)

\[\text{divisors n == b := factors(n :: FR) c := concat(1,"append")} \\
\left[\text{[a.factor**j for j in 1..a.exponent] for a in b]}\right] \\
\text{if #(b) = 1 then c else concat(n,c)}\]

ss: \( (I,I) \rightarrow SPOL RN \)

\[\text{ss(n,m) == li : L I := [n for j in 1..m]} \\
\text{monomial(1,partition li)}\]

\[\text{s n == ss(n,1)}\]

cyc n ==
\[n = 1 \Rightarrow s 1 \\
+([(\text{eulerPhi(i) / n) * ss(i,numer(n/i)) for i in divisors n}]\]

dih n ==
\[k := n \text{ quo 2} \\
\text{odd? n } \Rightarrow (1/2) \ * \ \text{cyc n} \ + \ (1/2) \ * \ \text{ss(2,k) } \ * \ s 1 \\
(1/2) \ * \ \text{cyc n} \ + \ (1/4) \ * \ \text{ss(2,k)} \ + \ (1/4) \ * \ \text{ss(2,k-1)} \ * \ ss(1,2)\]

trm2: \( L I \rightarrow SPOL RN \)

\[\text{trm2 li == lli := powers(li)$PTN xx := 1/(pdct partition li) prod : SPOL RN := 1 for li in lli repeat l10 := first li; l11 := second li k := l10 \text{ quo 2} c :=} \\
\text{odd? l10 } \Rightarrow ss(l10,l11 * k) \\
\text{ss(k,l11) * ss(l10,l11 * (k - 1))} \\
c := c * ss(l10,l10 * ((l11*(l11 - 1)) quo 2)) prod2 : SPOL RN := 1 for r in lli | first(r) < l10 repeat r0 := first r; r1 := second r prod2 := ss(lcm(r0,l10),gcd(r0,l10) * r1 * l11) * prod2 prod := c * prod2 * prod xx * prod}\]

\[\text{graphs n == +/}[\text{trm2 li for li in list(partitions n)]]\]

cupp: \( (PTN,SPOL RN) \rightarrow SPOL RN \)
cupp(pt,spol) ==
  zero? spol => 0
  (dg := degree spol) < pt => 0
  dg = pt => (pdct pt) * monomial(leadingCoefficient spol,dg)
  cupp(pt,eductum spol)

cup(spol1,spol2) ==
  zero? spol1 => 0
  p := leadingCoefficient(spol1) * cupp(degree spol1,spol2)
  p + cup(reductum spol1,spol2)
ev spol ==
  zero? spol => 0
  leadingCoefficient(spol) + ev(reductum spol)
cap(spol1,spol2) == ev cup(spol1,spol2)

mtpol: (I,SPOL RN) -> SPOL RN
mtpol(n,spol)==
  zero? spol => 0
  deg := partition [n*k for k in (degree spol)::L(I)]
  monomial(leadingCoefficient spol,deg) + mtpol(n,eductum spol)

fn2: I -> SPOL RN
evspol: ((I -> SPOL RN),SPOL RN) -> SPOL RN
evspol(fn2,spol) ==
  zero? spol => 0
  lc := leadingCoefficient spol
  prod := _*/[fn2 i for i in (degree spol)::L(I)]
  lc * prod + evspol(fn2,eductum spol)
wreath(spol1,spol2) == evspol(x+->mtpol(x,spol2),spol1)

hh: I -> SPOL RN --symmetric group
hh n == if n=0 then 1 else if n<0 then 0 else h n
SFunction li==
a:Matrix SPOL RN:=matrix [[hh(k -j+i) for k in li for j in 1..#li]
  for i in 1..#li]
  determinant a

roundup:(L I,L I)-> L I
roundup(li1,li2)==
  #li1 > #li2 => roundup(li1,concat(li2,0))
  li2
skewSFunction(li1,li2)==
  #li1 < #li2 =>
    error "skewSFunction: partition1 does not include partition2"
  li2:=roundup (li1,li2)
a:Matrix SPOL RN:=matrix [[hh(k-li2.i-j+i)
for k in li1 for j in 1..#li1 for i in 1..#li1

determinant a

---

— CYCLES.dotabb —

"CYCLES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLES"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLES" -> "PFECAT"

---

package CSTTOOLS CyclicStreamTools

— CyclicStreamTools.input —

)set break resume
)sys rm -f CyclicStreamTools.output
)spool CyclicStreamTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CyclicStreamTools
--E 1

)spool
)lisp (bye)

---

— CyclicStreamTools.help —

==============================================
CyclicStreamTools examples
==============================================

This package provides tools for working with cyclic streams.

See Also:
  o )show CyclicStreamTools

---
CyclicStreamTools (CSTTOOLS)

Exports:
computeCycleEntry computeCycleLength cycleElt

— package CSTTOOLS CyclicStreamTools —

)abbrev package CSTTOOLS CyclicStreamTools
++ Author: Clifton J. Williamson
++ Date Created: 5 December 1989
++ Date Last Updated: 5 December 1989
++ Description:
++ This package provides tools for working with cyclic streams.

CyclicStreamTools(S,ST): Exports == Implementation where
S : Type
ST : LazyStreamAggregate S

Exports ==> with

cycleElt: ST -> Union(ST,"failed")
  ++ cycleElt(s) returns a pointer to a node in the cycle if the stream
  ++ s is cyclic and returns "failed" if s is not cyclic
  ++
  ++X p:=repeating([1,2,3])
  ++X q:=cons(4,p)
  ++X cycleElt q
  ++X r:=[1,2,3]::Stream(Integer)
  ++X cycleElt r

computeCycleLength: ST -> NonNegativeInteger
  ++ computeCycleLength(s) returns the length of the cycle of a
++ cyclic stream t, where s is a pointer to a node in the  
++ cyclic part of t.
++
++ X p:=repeating([1,2,3])
++ X q:=cons(4,p)
++ X computeCycleLength(cycleElt(q))

computeCycleEntry: (ST,ST) -> ST
++ computeCycleEntry(x,cycElt), where cycElt is a pointer to a
++ node in the cyclic part of the cyclic stream x, returns a
++ pointer to the first node in the cycle
++
++ X p:=repeating([1,2,3])
++ X q:=cons(4,p)
++ X computeCycleEntry(q,cycleElt(q))

Implementation ==> add

cycleElt x ==
y := x
for i in 0.. repeat
  (explicitlyEmpty? y) or (lazy? y) => return "failed"
y := rst y
  if odd? i then x := rst x
  eq?(x,y) => return y

computeCycleLength cycElt ==
i : NonNegativeInteger
y := cycElt
for i in 1.. repeat
  y := rst y
  eq?(y,cycElt) => return i

computeCycleEntry(x,cycElt) ==
y := rest(x, computeCycleLength cycElt)
repeat
  eq?(x,y) => return x
  x := rst x ; y := rst y

— CSTTOOLS.dotabb —
"CSTTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CSTTOOLS"]
"LZSTAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LZSTAGG"]
"CSTTOOLS" -> "LZSTAGG"
package CYCLOTOM CyclotomicPolynomialPackage

--- CyclotomicPolynomialPackage.input ---

)set break resume
/sys rm -f CyclotomicPolynomialPackage.output
/spool CyclotomicPolynomialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CyclotomicPolynomialPackage
--E 1

)spool
)lisp (bye)

---

--- CyclotomicPolynomialPackage.help ---

====================================================================
CyclotomicPolynomialPackage examples
====================================================================

This package has no description

See Also:
  o )show CyclotomicPolynomialPackage

---
CyclotomicPolynomialPackage (CYCLOTOM)

Exports:
cyclotomic  cyclotomicDecomposition  cyclotomicFactorization

— package CYCLOTOM CyclotomicPolynomialPackage —

)abbrev package CYCLOTOM CyclotomicPolynomialPackage
++ Description:
++ This package has no description

CyclotomicPolynomialPackage: public == private where
  SUP ==> SparseUnivariatePolynomial(Integer)
  LSUP ==> List(SUP)
  NNI ==> NonNegativeInteger
  FR ==> Factored SUP
  IFP ==> IntegerFactorizationPackage Integer

public == with
  cyclotomicDecomposition: Integer -> LSUP
    ++ cyclotomicDecomposition(n) undocumented{}
  cyclotomic: Integer -> SUP
    ++ cyclotomic(n) undocumented{}
  cyclotomicFactorization: Integer -> FR
    ++ cyclotomicFactorization(n) undocumented{}

private == add
  cyclotomic(n:Integer): SUP ==
    x,y,z,l: SUP
    g := factors factor(n)$IFP
    --Now, for each prime in the factorization apply recursion
    l := monomial(1,1) - monomial(1,0)
    for u in g repeat
      l := (monicDivide(multiplyExponents(l,u.factor::NNI),l)).quotient
      if u.exponent>1 then
        l := multiplyExponents(l,((u.factor)**((u.exponent-1)::NNI)::NNI)::NNI)
    l
cyclo
tomic
Decomposition\(n: \text{Integer}\):\(\text{LSUP}\) ==
x,y,z: \text{SUP}
l,ll,m: \text{LSUP}
rr: \text{Integer}
g := \text{factors} \text{factor}(n)$\text{IFP}$
l := [\text{monomial}(1,1) - \text{monomial}(1,0)]
\text{--Now, for each prime in the factorization apply recursion}
for \(u\) in \(g\) repeat
\quad m := [(\text{monicDivide}(
\text{multiplyExponents}(z,u.\text{factor}::\text{NNI}),z)).\text{quotient} \text{for} z \text{in} l]
\quad \text{for} rr \text{in} 1..(u.\text{exponent}-1) \text{repeat}
\quad \quad l := \text{append}(l,m)
\quad m := [\text{multiplyExponents}(z,u.\text{factor}::\text{NNI}) \text{for} z \text{in} m]
\quad l := \text{append}(l,m)
l
\text{cyclo
tomicFactorization}\(n: \text{Integer}\):\(\text{FR}\) ==
f : \text{SUP}
fr : \text{FR} := 1$\text{FR}$
\text{for} f \text{in} \text{cyclo
tomicDecomposition}(n) \text{repeat}
\quad fr := fr * \text{primeFactor}(f,1$\text{Integer}$)
fr

| CYCLO
tOM.dotabb |

"CYCLO
tOM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLO
tOM"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLO
tOM" -> "PFECAT"
Chapter 5

Chapter D

package DFINTTLS DefiniteIntegrationTools

— DefiniteIntegrationTools.input —

)set break resume
)sys rm -f DefiniteIntegrationTools.output
)spool DefiniteIntegrationTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DefiniteIntegrationTools
--E 1

)spool
)lisp (bye)

— DefiniteIntegrationTools.help —

====================================================================
DefiniteIntegrationTools examples
====================================================================

DefiniteIntegrationTools provides common tools used by the definite integration of both rational and elementary functions.

See Also:

293
DefiniteIntegrationTools (DFINTTLS)

Exports:
ignore?  checkForZero  computeInt

— package DFINTTLS DefiniteIntegrationTools —

)abbrev package DFINTTLS DefiniteIntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 15 April 1992
++ Date Last Updated: 24 February 1993
++ Description:
++ \spadtype{DefiniteIntegrationTools} provides common tools used
++ by the definite integration of both rational and elementary functions.

DefiniteIntegrationTools(R, F): Exports == Implementation where
R : Join(GcdDomain, OrderedSet, RetractableTo Integer,
          LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory,
          AlgebraicallyClosedFunctionSpace R)
B  ==> Boolean
Z  ==> Integer
Q  ==> Fraction Z
SE ==> Symbol
P  ==> Polynomial R
RF ==> Fraction P
UP ==> SparseUnivariatePolynomial F
K  ==> Kernel F
OFE ==> OrderedCompletion F
UPZ ==> SparseUnivariatePolynomial Z
UPQ ==> SparseUnivariatePolynomial Q
REC ==> Record(left:Q, right:Q)
REC2==> Record(endpoint:Q, dir:Z)
U ==> Union(fin:REC, halfinf:REC2, all:"all", failed:"failed")
IGNOR ==> "noPole"

Exports ==> with
  ignore?: String -> B
    ++ ignore?(s) is true if s is the string that tells the integrator
    ++ to assume that the function has no pole in the integration interval.
  computeInt: (K, F, OFE, OFE, B) -> Union(OFE, "failed")
    ++ computeInt(x, g, a, b, eval?) returns the integral of \spad{f} for x
    ++ between a and b, assuming that g is an indefinite integral of
    ++ \spad{f} and \spad{f} has no pole between a and b.
    ++ If \spad{eval?} is true, then \spad{g} can be evaluated safely
    ++ at \spad{a} and \spad{b}, provided that they are finite values.
    ++ Otherwise, limits must be computed.
  checkForZero: (P, SE, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, x, a, b, incl?) is true if p has a zero for x between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.
  checkForZero: (UP, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, a, b, incl?) is true if p has a zero between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.

Implementation ==> add
  import RealZeroPackage UPZ
  import InnerPolySign(F, UP)
  import ElementaryFunctionSign(R, F)
  import PowerSeriesLimitPackage(R, F)
  import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)

  mkLogPos : F -> F
  keeprec? : (Q, REC) -> B
  negative : F -> Union(B, "failed")
  mkKerPos : K -> Union(F, "positive")
  posRoot : (UP, B) -> Union(B, "failed")
  realRoot : UP -> Union(B, "failed")
  var : UP -> Union(Z, "failed")
  maprat : UP -> Union(UPZ, "failed")
  variation : (UP, F) -> Union(Z, "failed")
  infeval : (UP, OFE) -> Union(F, "failed")
  checkHalfAx : (UP, F, Z, B) -> Union(B, "failed")
  findLimit : (F, K, OFE, String, B) -> Union(OFE, "failed")
  checkBudan : (UP, OFE, OFE, B) -> Union(B, "failed")
  checkDeriv : (UP, OFE, OFE) -> Union(B, "failed")
  sameSign : (UP, OFE, OFE) -> Union(B, "failed")
intrat : (OFE, OFE) -> U
findRealZero: (UPZ, U, B) -> List REC

variation(p, a) == var p(monomial(1, 1)$UP - a::UP)
keeprec?(a, rec) == (a > rec.right) or (a < rec.left)

checkHalfAx(p, a, d, incl?) ==
  posRoot(p(d * (monomial(1, 1)$UP - a::UP)), incl?)

ignore? str ==
  str = IGNOR => true
  error "integrate: last argument must be 'noPole'"

computeInt(k, f, a, b, eval?) ==
  if not eval? then f := mkLogPos f
  ((ib := findLimit(f, k, b, "left", eval?)) case "failed") or
  ((ia := findLimit(f, k, a, "right", eval?)) case "failed") => "failed"
  infinite?(ia::OFE) and (ia::OFE = ib::OFE) => "failed"
  ib::OFE - ia::OFE

findLimit(f, k, a, dir, eval?) ==
  r := retractIfCan(a)@Union(F, "failed")
  r case F =>
    eval? => mkLogPos(eval(f, k, r::F))::OFE
    (u := limit(f, equation(k::F, r::F), dir)) case OFE => u::OFE
    "failed"
  (u := limit(f, equation(k::F::OFE, a))) case OFE => u::OFE
  "failed"

mkLogPos f ==
  lk := empty()$List(K)
  lv := empty()$List(F)
  for k in kernels f | is?(k, "log"::SE) repeat
    if (v := mkKerPos k) case F then
      lk := concat(k, lk)
      lv := concat(v::F, lv)
    eval(f, lk, lv)

mkKerPos k ==
  (u := negative(f := first argument k)) case "failed" =>
    log(f**2) / (2::F)
  u::B => log(-f)
  "positive"

negative f ==
  (u := sign f) case "failed" => "failed"
  u::Z < 0

checkForZero(p, x, a, b, incl?) ==
checkForZero(
  map(s->s::F, univariate(p, x))
) $\text{SparseUnivariatePolynomialFunctions2}(P, F), a, b, \text{incl}?$

checkForZero(q, a, b, incl?) ==
  ground? q => false
  (d := maprat q) case UPZ and not((i := intrat(a, b)) case failed) =>
    not empty? findRealZero(d::UPZ, i, incl?)
  (u := checkBudan(q, a, b, incl?)) case "failed" =>
    incl? => checkDeriv(q, a, b)
    "failed"
  u::B

maprat p ==
  ans:UPQ := 0
  while p ^= 0 repeat
    (r := retractIfCan(c := leadingCoefficient p)@Union(Q, "failed"))
    case "failed" => return "failed"
    ans := ans + monomial(r::Q, degree p)
  p := reductum p
  map(numer,(splitDenominator ans).num)
) $\text{SparseUnivariatePolynomialFunctions2}(Q, Z)$

intrat(a, b) ==
  (n := whatInfinity a) ^= 0 =>
    (r := retractIfCan(b)@Union(F, "failed")) case "failed" => ["all"]
    (q := retractIfCan(r::F)@Union(Q, "failed")) case "failed" =>
      ["failed"]
      [[q::Q, n]]
  (q := retractIfCan(retract(a)@F)@Union(Q, "failed")) case "failed" =>
    ["failed"]
    [[q::Q, t::Q]]
  (t := retractIfCan(retract(b)@F)@Union(Q, "failed")) case "failed" =>
    ["failed"]
    [[q::Q, t::Q]]

findRealZero(p, i, incl?) ==
  i case fin =>
    l := realZeros(p, r := i.fin)
    incl? => l
    select_!(s+->keeprec?(r.left, s) and keeprec?(r.right, s), l)
  i case all => realZeros p
  i case halfinf =>
    empty?(l := realZeros p) => empty()
    bounds:REC :=
      i.halfinf.dir > 0 => [i.halfinf.endpoint, "max"/[t.right for t in l]]
      ["min"/[t.left for t in l], i.halfinf.endpoint]
    l := [u::REC for t in l | (u := refine(p, t, bounds)) case REC]
    incl? => l
ep := i.halfinf.endpoint
select_!(s->keeprec?(ep, s), 1)
error "findRealZero: should not happen"

checkBudan(p, a, b, incl?) ==
r := retractIfCan(b)@Union(F, "failed")
(n := whatInfinity a) ^= 0 =>
r case "failed" => realRoot p
checkHalfAx(p, r::F, n, incl?)
(za? := zero? p(aa := retract(a)@F)) and incl? => true
(n := whatInfinity b) ^= 0 => checkHalfAx(p, aa, n, incl?)
(zb? := zero? p(bb := r::F)) and incl? => true
(va := variation(p, aa)) case "failed" or
  (vb := variation(p, bb)) case "failed" => "failed"
m:Z := 0
if za? then m := inc m
if zb? then m := inc m
odd?(v := va::Z - vb::Z) =>
  -- p has an odd number of roots
  incl? or even? m => true
  -- one? v => false
  (v = 1) => false
  "failed"
zero? v => false
  -- p has no roots
  -- one? m => true -- p has an even number > 0 of roots
  (m = 1) => true
  "failed"

checkDeriv(p, a, b) ==
(r := retractIfCan(p)@Union(F, "failed")) case F => zero?(r::F)
(s := sameSign(p, a, b)) case "failed" => "failed"
s:B => -- p has the same nonzero sign at a and b
  (u := checkDeriv(differentiate p,a,b)) case "failed" => "failed"
u::B => "failed"
true

realRoot p ==
(b := posRoot(p, true)) case "failed" => "failed"
b::B => true
posRoot(p(p - monomial(1, 1)$UP), true)

sameSign(p, a, b) ==
(ea := infeval(p, a)) case "failed" => "failed"
(eb := infeval(p, b)) case "failed" => "failed"
(s := sign(ea::F * eb::F)) case "failed" => "failed"
s::Z > 0

-- returns true if p has a positive root. Include 0 is incl0? is true
posRoot(p, incl0?) ==
  (z0? := zero?(coefficient(p, 0))) and incl0? => true
package DEGRED DegreeReductionPackage

---

infeval(p, a) ==
  zero?(n := whatInfinity a) => p(retract(a)@F)
  (u := signAround(p, n, sign)) case "failed" => "failed"
u::Z::F

var q ==
i:Z := 0
  (lastCoef := negative leadingCoefficient q) case "failed" =>
  "failed"
while ((q := reductum q) ^= 0) repeat
  (next := negative leadingCoefficient q) case "failed" =>
  return "failed"
  if ((not(lastCoef::B)) and next::B) or
    ((not(next::B)) and lastCoef::B) then i := i + 1
  lastCoef := next
i

---

"DFINTTLS" []
"ACFS" []
"DFINTTLS" -> "ACFS"

---

package DEGRED DegreeReductionPackage

---

)set break resume
)sys rm -f DegreeReductionPackage.output
)spool DegreeReductionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DegreeReductionPackage
--E 1

)spool
)lisp (bye)

——

— DegreeReductionPackage.help —

====================================================================
DegreeReductionPackage examples
====================================================================

This package has no description

See Also:
o )show DegreeReductionPackage

——

DegreeReductionPackage (DEGRED)

Exports:
expand    reduce

—— package DEGRED DegreeReductionPackage ——
)abbrev package DEGRED DegreeReductionPackage
++ Description:
++ This package has no description

DegreeReductionPackage(R1, R2): Cat == Capsule where
  R1: Ring
  R2: Join(IntegralDomain,OrderedSet)

I ==> Integer
PI ==> PositiveInteger
UP ==> SparseUnivariatePolynomial
RE ==> Expression R2

Cat == with
  reduce: UP R1 -> Record(pol: UP R1, deg: PI)
  expand: (RE, PI) -> List RE

Capsule == add

degrees(u: UP R1): List Integer ==
  l: List Integer := []
  while u ^= 0 repeat
    l := concat(degree u,l)
    u := reductum u
  l

reduce(u: UP R1) ==
  g := "gcd"/[d for d in degrees u]
  u := divideExponents(u, g:PI)::(UP R1)
  [u, g:PI]

import Fraction Integer

rootOfUnity(j:I,n:I):RE ==
  j = 0 => 1
  arg:RE := 2*j*pi()/(n::RE)
  cos arg + (-1)**(1/2) * sin arg

expand(s, g) ==
  g = 1 => [s]
  [rootOfUnity(i,g)*s**(1/g) for i in 0..g-1]
package DTP DesingTreePackage

— DesingTreePackage.input —

)set break resume
)sys rm -f DesingTreePackage.output
)spool DesingTreePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DesingTreePackage

--R
--R DesingTreePackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: ... DesingTreeCategory(InfClsPoint),BLMET: BlowUpMethodCategory) is a package constructor

--R Abbreviation for DesingTreePackage is DTP

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DTP

--R

--R--------------------------------- Operations ---------------------------------

--R fullParamInit : DesTree -> Void
--R initParLocLeaves : DesTree -> Void
--R adjunctionDivisor : DesTree -> DIVISOR
--R blowUp : InfClsPoint -> List(InfClsPoint)
--R blowUpWithExcpDiv : DesTree -> Void

--R desingTree : PolyRing -> List(DesTree)

--R resinorAtDesingTree : (PolyRing,DesTree) -> DIVISOR

--R genus : PolyRing -> NonNegativeInteger

--R genusTree : (NonNegativeInteger,List(DesTree)) -> NonNegativeInteger

--R genusTreeNeg : (NonNegativeInteger,List(DesTree)) -> Integer

--R inBetweenExcpDiv : DesTree -> DIVISOR

--R initializeParamOfPlaces : DesTree -> Void

--R initializeParamOfPlaces : (DesTree,List(PolyRing)) -> Void

--R

--E 1

)spool
The following is all the categories, domains and package
used for the desingularisation be means of
monoidal transformation (Blowing-up)

See Also:
o )show DesingTreePackage

DesingTreePackage (DTP)
DesingTreePackage(K, symb, PolyRing, E, ProjPt, PCS, Plc, DIVISOR, InfClsPoint, DesTree, BLMET):Exports == Implementation where

K:Field
symb: List(Symbol)

OV ==> OrderedVariableList(symb)

E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)

bls ==> ['X,'Y]
BlUpRing ==> DistributedMultivariatePolynomial( bls , K)
E2 ==> DirectProduct( #bls , NonNegativeInteger )
AFP ==> AffinePlane(K)
OV2 ==> OrderedVariableList( bls )
PI ==> PositiveInteger
INT ==> Integer
NNI ==> NonNegativeInteger
LPARSPT ==> LocalParametrizationOfSimplePointPackage
PARAMP ==> ParametrizationPackage
PRALGPK ==> ProjectiveAlgebraicSetPackage

InfClsPoint : InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt,_,PCS,Plc,DIVISOR,BLMET)

DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory

PackPoly ==> PackageForPoly(K,PolyRing,E,#symb)
PACKBL ==> PackageForPoly( K , BlUpRing , E2 , #bls )
NP ==> NewtonPolygon(K,BlUpRing,E2,#bls)
PPFC1 ==> PolynomialPackageForCurve(K, PolyRing, E, #symb, ProjPt)
PPFC2 ==> BlowUpPackage(K, symb, PolyRing, E, BLMET)
ParamPackFC ==> LPARSPT(K, symb, PolyRing, E, ProjPt, PCS, Plc)
ParamPack ==> PARAMP(K, symb, PolyRing, E, ProjPt, PCS, Plc)
PrjAlgPack ==> PRALGPK(K, symb, PolyRing, E, ProjPt)

Exports ==> with

  blowUp: InfClsPoint -> List InfClsPoint

  divisorAtDesingTree: (PolyRing, DesTree) -> DIVISOR
  ++ divisorAtDesingTree(f, tr) computes the local
  ++ divisor of f at a desingularisation tree tr of
  ++ a singular point.

  adjunctionDivisor: DesTree -> DIVISOR
  ++ adjunctionDivisor(tr) compute the local
  ++ adjunction divisor of a desingularisation tree tr of
  ++ a singular point.

  blowUpWithExcpDiv: DesTree -> Void -- DesTree

  desingTreeAtPoint: (ProjPt, PolyRing) -> DesTree
  ++ desingTreeAtPoint(pt, pol) computes
  ++ the desingularisation tree at the point pt
  ++ on the curve defined by pol.
  ++ This function recursively compute the tree.

  desingTree: PolyRing -> List DesTree
  ++ desingTree(pol) returns all the desingularisation
  ++ trees of all singular points on the curve
  ++ defined by pol.

  fullParamInit: DesTree -> Void
  ++ fullParamInit(tr) initialize the local
  ++ parametrization at all places (leaves of tr),
  ++ computes the local exceptional divisor
  ++ at each infinitely close points in the tree.
  ++ This function is equivalent to the following called:
  ++ initParLocLeaves(tr)
  ++ initializeParamOfPlaces(tr)
  ++ blowUpWithExcpDiv(tr)

  initParLocLeaves: DesTree -> Void
  ++ initParLocLeaves(tr) initialize the local
  ++ parametrization at simple points corresponding to
  ++ the leaves of tr.

  initializeParamOfPlaces: DesTree -> Void
  ++ initParLocLeaves(tr) initialize the local
++ parametrization at places corresponding to
  ++ the leaves of tr.

initializeParamOfPlaces: (DesTree, List PolyRing) -> Void
  ++ initParLocLeaves(tr,listOfFnc) initialize
  ++ the local parametrization at places corresponding to
  ++ the leaves of tr according to the given
  ++ list of functions in listOfFnc.

genus: PolyRing -> NNI
  ++ genus(pol) computes the genus of the curve defined by pol.

genusNeg: PolyRing -> INT
  ++ genusNeg(pol) computes the "genus" of a curve
  ++ that may be not absolutely irreducible.
  ++ A "negative" genus means that
  ++ the curve is reducible !.

genusTree: (NNI, List(DesTree)) -> NNI
  ++ genusTree(n,listOfTrees) computes the genus of a curve,
  ++ where n is the degree of a polynomial pol
  ++ defining the curve and listOfTrees is all
  ++ the desingularisation trees at all singular points
  ++ on the curve defined by pol.

inBetweenExcpDiv: DesTree -> DIVISOR

genusTreeNeg: (NNI, List(DesTree)) -> INT
  ++ genusTreeNeg(n,listOfTrees) computes the "genus"
  ++ of a curve that may be not absolutely irreducible,
  ++ where n is the degree of a polynomial pol
  ++ defining the curve and listOfTrees is all the
  ++ desingularisation trees at all singular points
  ++ on the curve defined by pol.
  ++ A "negative" genus means that
  ++ the curve is reducible !.

Implementation ==> add
import PackPoly
import PPFC1
import PPFC2
import PolyRing
import DesTree

divisorAtDesingTreeLocal: (BlUpRing, DesTree) -> DIVISOR
polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing
makeMono: DesTree -> BlUpRing
inBetweenExcpDiv( tr )==
  -- trouve le diviseur exp. d’un pt inf voisin PRECEDENT !
  -- qV est egal a : 1 + nombre de fois que ce point est repete
  -- dans un chaine (le plus un correspond au point d’origine du
  -- point dont il est question ici.
  -- mp est la multiplicité du point.
  -- cette fonction n’est et ne peut etre qu’utiliser pour
  -- calculer le diviseur d’adjonction ( a cause du mp -1).
  noeud:= value tr
  chart:= chartV noeud
  qV:= quotValuation chart
  one? qV => 0$DIVISOR
  expDiv := divisorAtDesingTreeLocal(makeMono(tr),tr)
  mp:= degree expDiv
  ((qV - 1) * (mp -1)) *$DIVISOR expDiv

polyRingToBlUpRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  ll:= [ d.i for i in 1..3 | ^i = chartCoord(chart) ]
  e:= directProduct( vector( ll)$Vector(NNI) )$E2
  monomial(lc , e )$BlUpRing + polyRingToBlUpRing( reductum pol, chart )

affToProj(pt:AFP, chart:BLMET ):ProjPt==
  nV:= chartCoord chart
  d:List(K) := list(pt)$AFP
  ll:List K:=
  nV = 1 => [ 1$K , d.1 , d.2 ]
  nV = 2 => [ d.1 , 01$K , d.2 ]
  [d.1 , d.2 , 1 ]
  projectivePoint( ll )$ProjPt

biringToPolyRing: (BlUpRing, BLMET) -> PolyRing
biringToPolyRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  nV:= chartCoord chart
  ll:List NNI:=
  nV = 1 => [ 0$NNI , d.1 , d.2 ]
  nV = 2 => [ d.1 , 0$NNI , d.2 ]
  [d.1 , d.2 , 0$NNI ]
  e:= directProduct( vector( ll)$Vector(NNI) )$E
  monomial(lc , e )$PolyRing + biringToPolyRing( reductum pol, chart )

minus : (NNI,NNI) -> NNI
minus(a,b)==
d:=subtractIfCan(a,b)

  d case "failed" => error "cannot substract a-b if b>a for NNI"

  d

-- returns the exceptional coordinate function

makeExcpDiv: List DesTree -> DIVISOR

desingTreeAtPointLocal: InfClsPoint -> DesTree

subGenus: DesTree -> NNI

lVar: List PolyRing := _
  [monomial(1,index(i pretend PI)$OV,1)$PolyRing for i in 1..#symb]

divisorAtDesingTreeLocal(pol,tr)==
  -- BLMET has QuadraticTransform ; marche aussi avec
  -- Hamburger-Noether mais surement moins efficace
  noeud:=value(tr)
  pt:=localPointV(noeud)
  chart:= chartV noeud
  -- ram:= ramifMult chart -- ???
  -- new way to compute in order not to translate twice pol
  polTrans:BlUpRing:=translate(pol,list(pt)$AFP)$PACKBL
  multPol:=degreeOfMinimalForm(polTrans)
  chtr:=children(tr)
  parPol:PCS
  ord:Integer
  empty?(chtr) =>
    parPol:=parametrize(biringToPolyRing(pol,chartV(noeud))_,
      localParamV(noeud))$ParamPack
    ord:=order(parPol)$PCS
  ord * excpDivV(noeud) -- Note: le div excp est une fois la place.
  (multPol *$DIVISOR excpDivV(noeud)) +$DIVISOR _
    reduce("+",[divisorAtDesingTreeLocal(_,quadTransform(polTrans,multPol,(chartV(value(child)))),_ child)_
      for child in chtr])

desingTreeAtPointLocal(ipt) ==
  -- crb:PolyRing,pt:ProjPt,lstnV:List(INT),origPoint:ProjPt,actL:K)==
  -- peut etre est-il preferable, avant d'eclater, de tester
  -- si le point est simple avec les derives, et non
  -- verifier si le point est simple ou non apres translation.
  -- ???
  blbl:=blowUp ipt
  multPt:=multV ipt
  one?(multPt) =>
    tree( ipt )$DesTree
  subTree:List DesTree:= [desingTreeAtPointLocal( ipt ) for ipt in blbl]
tree( ipt, subTree )$DesTree

blowUp(ipt)==
crb:=curveV ipt
pt:= localPointV ipt
lstnV := chartV ipt  -- CHH no modif needed
actL:= actualExtensionV ipt
origPoint:= pointV ipt
blbl:=stepBlowUp(crb,pt,lstnV,actL) -- CHH no modif needed
multPt:=blbl.mult
sm:= blbl.subMult
-- la multiplicite et la frontiere du polygone de Newton (ou la forme
-- minimale selon BLMET) du point ipt est assigne par effet de bord !
setmult!(ipt,multPt)
setsubmult!(ipt, sm)
one?(multPt) => empty()
[create(origPoint,_
    rec(recTransStr),_
    rec(recPoint) ,_
    0,_
    rec(recChart),_
    0,
    0$DIVISOR,_
    rec(definingExtension),_
    new(I)$Symbol )$InfClsPoint for rec in blbl.blUpRec]

makeMono(arb)==
monomial(1,index(excepCoord(chartV(value(arb))) pretend PI)$OV2,_,
1)$BlUpRing

makeExcpDiv(lstSsArb)==
reduce("+",_
    [divisorAtDesingTreeLocal(makeMono(arb),arb) for arb in lstSsArb],0)

adjunctionDivisorForQuadTrans: DesTree -> DIVISOR
adjunctionDivisorForHamburgeNoether: DesTree -> DIVISOR

adjunctionDivisor( tr )==
    BLMET has QuadraticTransform => adjunctionDivisorForQuadTrans( tr )
    BLMET has HamburgerNoether =>  adjunctionDivisorForHamburgeNoether( tr )
    error _
    " The algorithm to compute the adjunction divisor is not defined for the blowing method you have chosen"

adjunctionDivisorForHamburgeNoether( tr )==
    noeud:=value tr
    chtr:= children tr
    empty?(chtr) => 0$DIVISOR  -- on suppose qu'un noeud sans feuille
    -- est une feuille, donc non singulier. !
    multPt:= multV noeud
    ( minus(multPt,1) pretend INT) *$DIVISOR excepDivV(noeud) +$DIVISOR _
reduce("+$",[inBetweenExcpDiv( arb ) for arb in chtr ]) +$DIVISOR _
reduce("+$",[adjunctionDivisorForHamburgerNoether(arb) for arb in chtr])

adjunctionDivisorForQuadTrans(tr)==
noeud:=value(tr)
chtr:=children(tr)
empty?(chtr) => 0$DIVISOR
multPt:=multV(noeud)
( minus(multPt,1) pretend INT) *$DIVISOR excpDivV(noeud) +$DIVISOR _
reduce("+$",[adjunctionDivisorForQuadTrans(child) for child in chtr])

divisorAtDesingTree( pol , tr)==
chart:= chartV value(tr)
pp:= polyRingToBlUpRing( pol, chart )
divisorAtDesingTreeLocal( pp, tr )

subGenus(tr)==
noeud:=value tr
mult:=multV(noeud)
chart := chartV noeud
empty?(chtr:=children(tr)) => 0 -- degree(noeud)* mult* minus(mult,1)
degree(noeud)* ( mult*minus( mult, 1 ) + subMultV( noeud ) ) +
reduce("+$",[subGenus(ch) for ch in chtr])

initializeParamOfPlaces(tr,lpol)==
noeud:=value(tr)
pt:=localPointV(noeud)
crb:=curveV(noeud)
chart:=chartV(noeud) -- CHH
nV:INT:=chartCoord chart
chtr:List DesTree:=children(tr)
plc:Plc
lParam:List PCS
dd:PositiveInteger:=degree noeud
lcoef:List K
lll:Integer
lParInf:List(PCS)
lpar:List PCS
empty?(chtr) =>
lPar:=localParamOfSimplePt( affToProj(pt, chart) , _
biringToPolyRing(crb, chart),nV)$ParamPackFC
setlocalParam!(noeud,lPar)
lParam:=[parametrize( f , lPar)$ParamPack for f in lpol]
plc:= create( symbNameV(noeud) )$Plc
setParameter!(plc,lParam)
setDegree!(plc,dd)
itALeaf!(plc)
setexcpDiv!(noeud, plc :: DIVISOR )
void()}

lpolTrans:List PolyRing:=_
[translateToOrigin( pol, affToProj(pt, chart), nV) for pol in lpol]
lpolBlUp: List PolyRing
chartBl: BLMET
for arb in chtr repeat
  chartBl := chartV value arb
  lpolBlUp := [applyTransform(pol, chartBl) for pol in lpolTrans]
  initializeParamOfPlaces(arb, lpolBlUp)
  void()

blowUpWithExcpDiv(tr: DesTree) ==
  noeud := value(tr)
  pt := localPointV(noeud)
  crb := curveV(noeud)
  chtr := List DesTree := children(tr)
  empty?(chtr) => void() -- tr
  for arb in chtr repeat
    blowUpWithExcpDiv(arb)
    setexcpDiv!(noeud, makeExcpDiv(chtr))
  void()

fullParamInit(tr) ==
  initializeParamOfPlaces(tr)
  blowUpWithExcpDiv(tr)
  void()

initializeParamOfPlaces(tr) == initializeParamOfPlaces(tr, lVar)

desingTreeAtPoint(pt, crb) ==
  ipt := create(point, crb)$InfClsPoint
  desingTreeAtPointLocal ipt

genus(crb) ==
  if BLMET has HamburgerNoether then _
    print(" BUG BUG corige le bug GH ---- ")::OutputForm
  degCrb := totalDegree(crb)$PackPoly
  genusTree(degCrb, desingTree(crb))

genusNeg(crb) ==
  degCrb := totalDegree(crb)$PackPoly
  genusTreeNeg(degCrb, desingTree(crb))

desingTree(crb) ==
  desingTreeAtPoint(pt, crb) for pt in singularPoints(crb)$PrjAlgPack

genusTree(degCrb, listArbDes) ==
  -- le test suivant est necessaire
  -- ( meme s'il n'y a pas de point singulier dans ce cas)
  -- car avec sousNNI on ne peut retourner un entier negatif
  (degCrb <$NNI 3::NNI) and ^empty?(listArbDes) =>
    print("Too many infinitely near points")::OutputForm
print("The curve may not be absolutely irreducible")::OutputForm
error "Have a nice day"
(degCrb <$NNI 3::NNI) => 0
ga := ( minus(degCrb,1)*minus(degCrb ,2) ) quo$NNI 2
empty?(listArbDes) => ga
--calcul du nombre de double point
dp := reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2
(dp >$NNI ga) =>
  print("Too many infinitly near points")::OutputForm
  print("The curve may not be absolutely irreducible")::OutputForm
  error "Have a nice day"
minus(ga,dp)

genusTreeNeg(degCrb,listArbDes) =
  -- (degCrb <$NNI 3::NNI) => 0
ga := (degCrb-1)*(degCrb-2) quo$INT 2
empty?(listArbDes) => ga
ga- ( reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2)::INT

——

— DTP.dotabb —

"DTP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DTP"]
"INFCLCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=INFCLCT"]
"DTP" -> "INFCLCT"

——

package DIOSP DiophantineSolutionPackage

— DiophantineSolutionPackage.input —

)set break resume
)sys rm -f DiophantineSolutionPackage.output
)spool DiophantineSolutionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DiophantineSolutionPackage
--E 1
Any solution of a homogeneous linear Diophantine equation can be represented as a sum of minimal solutions, which form a "basis" (a minimal solution cannot be represented as a nontrivial sum of solutions) in the case of an inhomogeneous linear Diophantine equation, each solution is the sum of a inhomogeneous solution and any number of homogeneous solutions therefore, it suffices to compute two sets:

1. all minimal inhomogeneous solutions
2. all minimal homogeneous solutions

the algorithm implemented is a completion procedure, which enumerates all solutions in a recursive depth-first-search it can be seen as finding monotone paths in a graph for more details see Reference

See Also:
  o )show DiophantineSolutionPackage

---

DiophantineSolutionPackage (DIOSP)
Exports:
dioSolve

--- package DIOSP DiophantineSolutionPackage ---

)abbrev package DIOSP DiophantineSolutionPackage
++ Author: A. Fortenbacher
++ Date Created: 29 March 1991
++ Date Last Updated: 29 March 1991
++ Reference:
++  M. Clausen, A. Fortenbacher: Efficient Solution of
++  Linear Diophantine Equations. in JSC (1989) 8, 201-216
++ Description:
++  Any solution of a homogeneous linear Diophantine equation
++  can be represented as a sum of minimal solutions, which
++  form a "basis" (a minimal solution cannot be represented
++  as a nontrivial sum of solutions)
++  in the case of an inhomogeneous linear Diophantine equation,
++  each solution is the sum of an inhomogeneous solution and
++  any number of homogeneous solutions
++  therefore, it suffices to compute two sets:
++  \tab{5}1. all minimal inhomogeneous solutions
++  \tab{5}2. all minimal homogeneous solutions
++ the algorithm implemented is a completion procedure, which
++ enumerates all solutions in a recursive depth-first-search
++ it can be seen as finding monotone paths in a graph
++ for more details see Reference

DiophantineSolutionPackage(): Cat == Capsule where

B ==> Boolean
I ==> Integer
NI ==> NonNegativeInteger

LI ==> List(I)
VI ==> Vector(I)
VNI ==> Vector(NI)
POLI ==> Polynomial(I)
EPOLI ==> Equation(POLI)
LPOLI ==> List(POLI)

S ==> Symbol
LS ==> List(S)

ListSol ==> List(VNI)
Solutions ==> Record(varOrder: LS, inhom: Union(ListSol,"failed"),
                        hom: ListSol)
Node ==> Record(vert: VI, free: B)
Graph ==> Record(vn: Vector(Node), dim : NI, zeroNode: I)

Cat ==> with
dioSolve: EPOLI -> Solutions
  ++ dioSolve(u) computes a basis of all minimal solutions for
  ++ linear homogeneous Diophantine equation u,
  ++ then all minimal solutions of inhomogeneous equation

Capsule ==> add

import I
import POLI

-- local function specifications
initializeGraph: (LPOLI, I) -> Graph
createNode: (I, VI, NI, I) -> Node
findSolutions: (VNI, I, I, I, Graph, B) -> ListSol
verifyMinimality: (VNI, Graph, B) -> B
verifySolution: (VNI, I, I, I, Graph) -> B

-- exported functions
dioSolve(eq) ==
p := lhs(eq) - rhs(eq)
n := totalDegree(p)
n = 0 or n > 1 =>
  error "a linear Diophantine equation is expected"
mon := empty()$LPOLI
c : I := 0
for x in monomials(p) repeat
  ground?(x) =>
    c := ground(x) :: I
    mon := cons(x, mon)$LPOLI
graph := initializeGraph(mon, c)
sol := zero(graph.dim)$VNI
hs := findSolutions(sol, graph.zeroNode, 1, 1, graph, true)
ihs : ListSol :=
c = 0 => [sol]
findSolutions(sol, graph.zeroNode + c, 1, 1, graph, false)
vars := [first(variables(x))$LS for x in mon]
[vars, if empty?(ihs)$ListSol then "failed" else ihs, hs]

-- local functions
initializeGraph(mon, c) ==
  coeffs := vector([first(coefficients(x))$LI for x in mon])$VI
  k := #coeffs
m := min(c, reduce(min, coeffs)$VI)
n := max(c, reduce(max, coeffs)$VI)
[[createNode(i, coeffs, k, 1 - m) for i in m..n], k, 1 - m]

createNode(ind, coeffs, k, zeroNode) ==
-- create vertices from node ind to other nodes
v := zero(k)$VI
for i in 1..k repeat
  ind > 0 =>
    coeffs.i < 0 =>
      v.i := zeroNode + ind + coeffs.i
    coeffs.i > 0 =>
      v.i := zeroNode + ind + coeffs.i
  [v, true]

findSolutions(sol, ind, m, n, graph, flag) ==
-- return all solutions (paths) from node ind to node zeroNode
sols := empty()$ListSol
node := graph.vn.ind
node.free =>
  node.free := false
v := node.vert
k := if ind < graph.zeroNode then m else n
for i in k..graph.dim repeat
  x := sol.i
  v.i > 0 => -- vertex exists to other node
    sol.i := x + 1
  v.i = graph.zeroNode => -- solution found
    verifyMinimality(sol, graph, flag) =>
      sols := cons(copy(sol)$VNI, sols)$ListSol
    sol.i := x
  s :=
  ind < graph.zeroNode =>
    findSolutions(sol, v.i, i, n, graph, flag)
  findSolutions(sol, v.i, m, i, graph, flag)
  sols := append(s, sols)$ListSol
  sol.i := x
node.free := true
sols
sols

verifyMinimality(sol, graph, flag) ==
-- test whether sol contains a minimal homogeneous solution
flag => -- sol is a homogeneous solution
  i := 1
  while sol.i = 0 repeat
    i := i + 1
  x := sol.i
  sol.i := (x - 1) :: NI
flag := verifySolution(sol, graph.zeroNode, 1, 1, graph)
sol.i := x
flag
verifySolution(sol, graph.zeroNode, 1, 1, graph)

verifySolution(sol, ind, m, n, graph) ==
-- test whether sol contains a path from ind to zeroNode
flag := true
node := graph.vn.ind
v := node.vert
k := if ind < graph.zeroNode then m else n
for i in k..graph.dim while flag repeat
  x := sol.i
  x > 0 and v.i > 0 => -- vertex exists to other node
    sol.i := (x - 1) :: NI
    v.i = graph.zeroNode => -- solution found
      flag := false
      sol.i := x
  flag :=
  ind < graph.zeroNode =>
    verifySolution(sol, v.i, i, n, graph)
    verifySolution(sol, v.i, m, i, graph)
sol.i := x
flag

——

— DIOSP.dotabb —

"DIOSP" [color=#FF4488,href="bookvol10.4.pdf#nameddest=DIOSP"]
"IVECTOR" [color=#88FF44,href="bookvol10.3.pdf#nameddest=IVECTOR"]
"DIOSP" -> "IVECTOR"

——

package DIRPROD2 DirectProductFunctions2

—— DirectProductFunctions2.input ——

)set break resume
)sys rm -f DirectProductFunctions2.output
)spool DirectProductFunctions2.output
)set message test on
)set message auto off
This package provides operations which all take as arguments direct products of elements of some type \( A \) and functions from \( A \) to another type \( B \). The operations all iterate over their vector argument and either return a value of type \( B \) or a direct product over \( B \).

See Also:
- \( \text{show DirectProductFunctions2} \)
- \( \text{show DirectProductFunctions2} \)

Exports:
- map
- reduce
- scan
)abbrev package DIRPROD2 DirectProductFunctions2
++ Description:
++ This package provides operations which all take as arguments direct
++ products of elements of some type \spad{A} and functions from \spad{A}
++ to another type \spad{B}. The operations all iterate over their vector argument
++ and either return a value of type \spad{B} or a direct product over \spad{B}.

DirectProductFunctions2(dim, A, B): Exports == Implementation where
   dim : NonNegativeInteger
   A, B: Type
   DA ==> DirectProduct(dim, A)
   DB ==> DirectProduct(dim, B)
   VA ==> Vector A
   VB ==> Vector B
   O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)

Exports ==> with
   scan : ((A, B) -> B, DA, B) -> DB
   reduce : ((A, B) -> B, DA, B) -> B
   map : (A -> B, DA) -> DB

Implementation ==> add
   import FiniteLinearAggregateFunctions2(A, VA, B, VB)
   map(f, v) == directProduct map(f, v::VA)
   scan(f, v, b) == directProduct scan(f, v::VA, b)
   reduce(f, v, b) == reduce(f, v::VA, b)

— DIRPROD2.dotabb —

"DIRPROD2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DIRPROD2"]
"VECTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=VECTCAT"]
"DIRPROD2" -> "VECTCAT"
package DLP DiscreteLogarithmPackage

— DiscreteLogarithmPackage.input —

)set break resume
)sys rm -f DiscreteLogarithmPackage.output
)spool DiscreteLogarithmPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DiscreteLogarithmPackage
--E 1

)spool
)lisp (bye)

— DiscreteLogarithmPackage.help —

====================================================================
DiscreteLogarithmPackage examples
====================================================================

DiscreteLogarithmPackage implements help functions for discrete logarithms
in monoids using small cyclic groups.

See Also:
  o )show DiscreteLogarithmPackage
DiscreteLogarithmPackage (DLP)

Exports:
shanksDiscLogAlgorithm

— package DLP DiscreteLogarithmPackage —

)abbrev package DLP DiscreteLogarithmPackage
++ Author: J. Grabmeier, A. Scheerhorn
++ Date Created: 12 March 1991
++ Date Last Updated: 31 March 1991
++ References:
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ DiscreteLogarithmPackage implements help functions for discrete logarithms
++ in monoids using small cyclic groups.

DiscreteLogarithmPackage(M): public == private where
  M : Join(Monoid,Finite) with
    "**": (M,Integer) -> M
    ++ x ** n returns x raised to the integer power n
  public ==> with
    shanksDiscLogAlgorithm:(M,M,NonNegativeInteger)-> Union(NonNegativeInteger, "failed")
    ++ shanksDiscLogAlgorithm(b,a,p) computes s with \spad{b**s = a} for
    ++ assuming that \spad{a} and b are elements in a 'small' cyclic group of
    ++ order p by Shank's algorithm.
    ++ Note that this is a subroutine of the function \spadfun{discreteLog}.
  I  ==> Integer
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  SUP ==> SparseUnivariatePolynomial
  DLP ==> DiscreteLogarithmPackage

private ==> add
  shanksDiscLogAlgorithm(logbase,c,p) ==
limit:Integer:= 30
-- for logarithms up to cyclic groups of order limit a full
-- logarithm table is computed
p < limit =>
a:M:=1
disclog:Integer:=0
found:Boolean:=false
for i in 0..p-1 while not found repeat
  a = c =>
    disclog:=i
    found:=true
    a:=a*logbase
  not found =>
    messagePrint("discreteLog: second argument not in cyclic group_
generated by first argument")$OutputForm
    "failed"
    disclog pretend NonNegativeInteger
l:Integer:=length(p)$Integer
if odd?(l)$Integer then n:Integer:= shift(p,-(l quo 2))
  else n:Integer:= shift(1,(l quo 2))
a:M:=1
exptable : Table(PI,NNI) :=table()$Table(PI,NNI)
for i in (0::NNI)..(n-1)::NNI repeat
  insert_!([lookup(a),i::NNI]$Record(key:PI,entry:NNI),_
    exptable)$Table(PI,NNI)
a:=a*logbase
found := false
end := (p-1) quo n
disclog:Integer:=0
a := c
b := logbase ** (-n)
for i in 0..end while not found repeat
  rho:= search(lookup(a),exptable)_
    $Table(PositiveInteger,NNI)
rho case NNI =>$
    found := true
    disclog:= n * i + rho pretend Integer
    a := a * b
  not found =>
    messagePrint("discreteLog: second argument not in cyclic group_
generated by first argument")$OutputForm
    "failed"
    disclog pretend NonNegativeInteger

— DLP.dotabb —
package DISPLAY DisplayPackage

— DisplayPackage.input —

)set break resume
)sys rm -f DisplayPackage.output
)spool DisplayPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DisplayPackage
--E 1

)spool
)lisp (bye)

— DisplayPackage.help —

====================================================================
DisplayPackage examples
====================================================================

DisplayPackage allows one to print strings in a nice manner, including highlighting substrings.

See Also:
 o )show DisplayPackage


DisplayPackage (DISPLAY)

Exports:
  bright  center  copies  newLine  say  sayLength

— package DISPLAY DisplayPackage —

)abbrev package DISPLAY DisplayPackage
++ Author: Robert S. Sutor
++ Date Created: September 1986
++ Description:
++ DisplayPackage allows one to print strings in a nice manner,
++ including highlighting substrings.

DisplayPackage: public == private where
  I  ==> Integer
  L  ==> List
  S  ==> String
  RECLR ==> Record(lhs : S, rhs : S)

public == with
  bright:  S    ->   L S
    ++ bright(s) sets the font property of the string s to bold-face type.
  bright:  L S  ->   L S
    ++ bright(l) sets the font property of a list of strings, l, to
    ++ bold-face type.
  newLine: ()   ->  S
    ++ newLine() sends a new line command to output.

  copies:  (I,S)  ->  S
    ++ copies(i,s) will take a string s and create a new string composed of
    ++ i copies of s.
  center:   (S,I,S)  ->  S
    ++ center(s,i,s) takes the first string s, and centers it within a string
    ++ of length i, in which the other elements of the string are composed
    ++ of as many replications as possible of the second indicated string, s
    ++ which must have a length greater than that of an empty string.
center: (L S, I, S) -> L S
++ center(l, i, s) takes a list of strings l, and centers them within a
++ list of strings which is i characters long, in which the remaining
++ spaces are filled with strings composed of as many repetitions as
++ possible of the last string parameter s.

say: S -> Void
++ say(s) sends a string s to output.
say: L S -> Void
++ say(l) sends a list of strings l to output.
sayLength: S -> I
++ sayLength(s) returns the length of a string s as an integer.
sayLength: L S -> I
++ sayLength(l) returns the length of a list of strings l as an integer.

private == add
--StringManipulations()

center0: (I, I, S) -> RECLR
s : S
l : L S
HION : S := "%b"
HIOFF : S := "%d"
NEWLINE : S := "%l"

bright s == [HION,s,HIOFF]$(L S)
bright l == cons(HION,append(l,list HIOFF))
newLine() == NEWLINE

copies(n : I, s : S) ==
n < 1 => ""
n = 1 => s
t : S := copies(n quo 2, s)
odd? n => concat [s,t,t]
concat [t,t]

center0(len : I, wid : I, fill : S) : RECLR ==
(wid < 1) or (len >= wid) => ["","]$RECLR
m : I := (wid - len) quo 2
t : S := copies(1 + (m quo (sayLength fill)),fill)
[t(1..m),t(1..wid-len-m)]$RECLR

center(s, wid, fill) ==
wid < 1 => ""
len : I := sayLength s
len = wid => s
len > wid => s(1..wid)
rec : RECLR := center0(len,wid,fill)
concat [rec.lhs,s,rec.rhs]

center(l, wid, fill) ==
  wid < 1 => ["\""]$(L S)
  len : I := sayLength l
  len = wid => l
  -- len > wid => s(1..wid)
  rec : RECLR := center0(len,wid,fill)
  cons(rec.lhs,append(l,list rec.rhs))

say s ==
  sayBrightly$Lisp s
  void()$Void

say l ==
  sayBrightly$Lisp l
  void()$Void

sayLength s == #s

sayLength l ==
  sum : I := 0
  for s in l repeat
    s = HION  => sum := sum + 1
    s = HIOFF => sum := sum + 1
    s = NEWLINE => sum
    sum := sum + sayLength s
  sum

display

package DDFACT DistinctDegreeFactorize

  — DistinctDegreeFactorize.input —

)set break resume
DistinctDegreeFactorize (DDFACT)

Package for the factorization of a univariate polynomial with coefficients in a finite field. The algorithm used is the "distinct degree" algorithm of Cantor-Zassenhaus, modified to use trace instead of the norm and a table for computing Frobenius as suggested by Naudin and Quitte.

See Also:
o )show DistinctDegreeFactorize
CHAPTER 5. CHAPTER D

Exports:

\begin{verbatim}
distdfact  exptMod  factor  factorSquareFree  irreducible?
separateDegrees  separateFactors  trace2PowMod  tracePowMod
\end{verbatim}

— package DDFACT DistinctDegreeFactorize —

\begin{verbatim}
)abbrev package DDFACT DistinctDegreeFactorize
++ Author: P. Gianni, B. Trager
++ Date Created: 1983
++ Date Last Updated: 22 November 1993
++ Description:
++ Package for the factorization of a univariate polynomial with
++ coefficients in a finite field. The algorithm used is the
++ "distinct degree" algorithm of Cantor-Zassenhaus, modified
++ to use trace instead of the norm and a table for computing
++ Frobenius as suggested by Naudin and Quitte.

DistinctDegreeFactorize(F,FP): C == T
where
F : FiniteFieldCategory
FP : UnivariatePolynomialCategory(F)
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FFE ==> Record(flg:fUnion, fctr:FP, xpnt:Integer)
NNI == NonNegativeInteger
Z == Integer
fact == Record(deg : NNI, prod : FP)
ParFact == Record(irr:FP, pow:Z)
FinalFact == Record(cont:F, factors:List(ParFact))

C == with
factor : FP -> Factored FP
++ factor(p) produces the complete factorization of the polynomial p.
factorSquareFree : FP -> Factored FP
++ factorSquareFree(p) produces the complete factorization of the
++ square free polynomial p.
distdfact : (FP,Boolean) -> FinalFact
++ distdfact(p, sqfrflag) produces the complete factorization
++ of the polynomial p returning an internal data structure.
++ If argument sqfrflag is true, the polynomial is assumed square free.
separateDegrees : FP -> List fact
++ separateDegrees(p) splits the square free polynomial p into
++ factors each of which is a product of irreducibles of the
++ same degree.
separateFactors : List fact -> List FP
++ separateFactors(lfact) takes the list produced by separateDegrees
++ and produces the complete list of factors.
exptMod : (FP,NNI,FP) -> FP
++ exptMod(u,k,v) raises the polynomial u to the kth power
++ modulo the polynomial v.
\end{verbatim}
trace2PowMod : (FP,NNI,FP) -> FP  
++ trace2PowMod(u,k,v) produces the sum of u**(2**i) for i running  
++ from 1 to k all computed modulo the polynomial v.

tracePowMod : (FP,NNI,FP) -> FP  
++ tracePowMod(u,k,v) produces the sum of \spad{u**(q**i)}  
++ for i running and q= size F

irreducible? : FP -> Boolean  
++ irreducible?(p) tests whether the polynomial p is irreducible.

T == add  
--declarations  
D:=ModMonic(F,FP)  
import UnivariatePolynomialSquareFree(F,FP)

--local functions  
notSqFr : (FP,FP -> List(FP)) -> List(ParFact)  
ddffact : FP -> List(FP)  
ddffact1 : (FP,Boolean) -> List fact  
ranpol : NNI -> FP

charF : Boolean := characteristic()$F = 2

--construct a random polynomial of random degree < d  
ranpol(d:NNI):FP ==  
k1: NNI := 0  
while k1 = 0 repeat k1 := random d  
-- characteristic F = 2  
charF =>  
  u:=0$FP  
  for j in 1..k1 repeat u:=u+monomial(random()$F,j)  
  u  
  u := monomial(1,k1)  
  for j in 0..k1-1 repeat u:=u+monomial(random()$F,j)  
  u

notSqFr(m:FP,appl: FP->List(FP)):List(ParFact) ==  
  factlist : List(ParFact) :=empty()  
  llf : List FFE  
  fln :List(FP) := empty()  
  if (lcm:=leadingCoefficient m)^=1 then m:=(inv lcm)*m  
  llf:= factorList(squareFree(m))  
  for lf in llf repeat  
    d1:= lf.xpnt  
    pol := lf.fctr  
    if (lcp:=leadingCoefficient pol)^=1 then pol := (inv lcp)*pol  
    degree pol=1 => factlist:=cons([pol,d1]$ParFact,factlist)  
    fln := appl(pol)  
    factlist :=append([[pf,d1]$ParFact for pf in fln],factlist)  
  factlist
-- compute $u^k \mod v$ (requires call to setPoly of multiple of $v$)
-- characteristic not equal 2
exptMod(u:FP,k:NNI,v:FP):FP == (reduce(u)$D**k):FP rem v

-- compute $u^k \mod v$ (requires call to setPoly of multiple of $v$)
-- characteristic equal 2
trace2PowMod(u:FP,k:NNI,v:FP):FP ==
  uu:=u
  for i in 1..k repeat uu:=(u+uu*uu) rem v
  uu

-- compute $u+u^q+..+u^{(q^k)} \mod v$ where $q=$size$< F$
tracePowMod(u:FP,k:NNI,v:FP):FP ==
  u1 :D :=reduce(u)$D
  uu : D := u1
  for i in 1..k repeat uu:=(u1+frobenius uu)
  (lift uu) rem v

-- compute $u^{(1+q+..+q^k)} \mod v$ where $q=#F$
-- frobenius map is used
normPowMod(u:FP,k:NNI,v:FP):FP ==
  u1 :D :=reduce(u)$D
  uu : D := u1
  for i in 1..k repeat uu:=(u1*frobenius uu)
  (lift uu) rem v

--find the factorization of $m$ as product of factors each containing
--terms of equal degree .
-- if testirr=true the function returns the first factor found
ddffact1(m:FP,testirr:Boolean):List(fact) ==
p:=size$F
  dg:NNI :=0
ddfact:List(fact):=empty()
  --evaluation of $x^p \mod m$
k1:NNI
  u:= m
  du := degree u
  setPoly u
  mon: FP := monomial(1,1)
  v := mon
  for k1 in 1.. while k1 <= (du quo 2) repeat
    v := lift frobenius reduce(v)$D
    g := gcd(v-mon,u)
    dg := degree g
    dg =0 => "next k1"
    if leadingCoefficient g ^=1 then g := (inv leadingCoefficient g)*g
    ddfact := cons([k1,g]$fact,ddfact)
-- test irreducibility
irreducible?(m:FP):Boolean ==
    mf:fact:=first ddfact1(m,true)
    degree m = mf.deg

--export ddfact1
separateDegrees(m:FP):List(fact) == ddfact1(m,false)

--find the complete factorization of m, using the result of ddfact1
separateFactors(distf : List fact) :List FP ==
    ddfact := distf
    n1:Integer
    p1:=size()$F
    if charF then n1:=length(p1)-1
    newaux,aux,ris : List FP
    ris := empty()
    t,fprod : FP
    for ffprod in ddfact repeat
        fprod := ffprod.prod
        d := fprod.deg
        degree fprod = d => ris := cons(fprod,ris)
        aux:=[fprod]
        setPoly fprod
        while ~(empty? aux) repeat
            t := ranpol(2*d)
            if charF then t:=trace2PowMod(t,(n1*d-1)::NNI,fprod)
            else t:=exptMod(tracePowMod(t,(d-1)::NNI,fprod),
                             (p1 quo 2)::NNI,fprod)-1$FP
            newaux:=empty()
            for u in aux repeat
                g := gcd(u,t)
                dg:= degree g
                dg=0 or dg = degree u =
                newaux:=cons(u,newaux)
                v := u quo g
                if dg=d then ris := cons(inv(leadingCoefficient g)*g,ris)
                else newaux := cons(g,newaux)
                if degree v=d then ris := cons(inv(leadingCoefficient v)*v,ris)
                else newaux := cons(v,newaux)
                aux:=newaux
            ris

--distinct degree algorithm for monic ,square-free polynomial
ddffact(m:FP):List(FP)==
ddfact := ddfact1(m, false)
empty? ddfact => [m]
separateFactors ddfact

--factorize a general polynomial with distinct degree algorithm
--if test=true no check is executed on square-free
distdfact(m: FP, test: Boolean): FinalFact ==
  factlist: List(ParFact) := empty()
  fln: List(FP) := empty()

  -- make m monic
  if (lcm := leadingCoefficient m) ^= 1 then m := (inv lcm)*m

  -- is x**d factor of m?
  if (d := minimumDegree m) > 0 then
    m := (monicDivide (m, monomial(1, d))).quotient
    factlist := [\[monomial(1, 1), d\]$ParFact]
    d := degree m

  -- is m constant?
  d = 0 => [lcm, factlist]$FinalFact

  -- is m linear?
  d = 1 => [lcm, cons([m, d]$ParFact, factlist)]$FinalFact

  -- m is square-free
  test =>
  fln := ddfact m
  factlist := append([\[pol, 1\]$ParFact for pol in fln], factlist)
  [lcm, factlist]$FinalFact

  -- factorize the monic, square-free terms
  factlist := append(notSqFr(m, ddfact), factlist)
  [lcm, factlist]$FinalFact

-- factorize the polynomial m
factor(m: FP) ==
  m = 0 => 0
  flist := distdfact(m, false)
  makeFR(flist.cont:: FP, [["prime", u.irr, u.pow]$FFE
    for u in flist.factors])

-- factorize the square free polynomial m
factorSquareFree(m: FP) ==
  m = 0 => 0
  flist := distdfact(m, true)
  makeFR(flist.cont:: FP, [["prime", u.irr, u.pow]$FFE
    for u in flist.factors])
The special functions in this section are developed as special cases but can all be expressed in terms of generalized hypergeometric functions $pF_q$ or its generalization, the Meijer G function. The long term plan is to reimplement these functions using the generalized version.

---

package DFSFUN DoubleFloatSpecialFunctions

---

--- DoubleFloatSpecialFunctions.input ---
--R En : (Integer,DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Gamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyAi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyAi : DoubleFloat -> DoubleFloat
--R airyBi : DoubleFloat -> DoubleFloat
--R airyBi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R besselI : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselI : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselJ : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselJ : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselK : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselK : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselY : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselY : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R digamma : DoubleFloat -> DoubleFloat
--R digamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R hypergeometric0F1 : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R hypergeometric0F1 : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R logGamma : DoubleFloat -> DoubleFloat
--R logGamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R polygamma : (NonNegativeInteger,DoubleFloat) -> DoubleFloat
--R polygamma : (NonNegativeInteger,Complex(DoubleFloat)) -> Complex(DoubleFloat)

pearceyC:=

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c
\end{array}
\]

[0.00, 0.0000000], [0.25, 0.3964561], [0.50, 0.5502472], [0.75, 0.6531193],
[1.00, 0.7217059], [1.25, 0.762404], [1.50, 0.779084], [1.75, 0.774978],
[2.00, 0.753302], [2.25, 0.717446], [2.50, 0.670986], [2.75, 0.617458],
[3.00, 0.561020], [3.25, 0.504745], [3.50, 0.452047], [3.75, 0.405762],
[4.00, 0.368193], [4.25, 0.341021], [4.50, 0.325249], [4.75, 0.321186],
[5.00, 0.328457], [5.25, 0.346058], [5.50, 0.372439], [5.75, 0.405610],
[6.00, 0.443274], [6.25, 0.482966], [6.50, 0.522202], [6.75, 0.558620],
[7.00, 0.590116], [7.25, 0.614951], [7.50, 0.631845], [7.75, 0.640034],
[8.00, 0.639301], [8.25, 0.629969], [8.50, 0.612868], [8.75, 0.589271],
[9.00, 0.560804], [9.25, 0.529344], [9.50, 0.496896], [9.75, 0.465469],
[10.00, 0.436964], [10.25, 0.413053], [10.50, 0.395087], [10.75, 0.384027],
[11.00, 0.380390], [11.25, 0.384231], [11.50, 0.395149], [11.75, 0.412319],
[12.00, 0.434555], [12.25, 0.460384], [12.50, 0.488146], [12.75, 0.516096],
[13.00, 0.542511], [13.25, 0.565798], [13.50, 0.584583], [13.75, 0.597795],
[14.00, 0.604721], [14.25, 0.605048], [14.50, 0.598871], [14.75, 0.586682],
[15.00, 0.569335], [15.25, 0.547984], [15.50, 0.524009], [15.75, 0.498930],
[16.00, 0.474310], [16.25, 0.451659], [16.50, 0.432343], [16.75, 0.417020],
[17.00, 0.407985], [17.25, 0.404300], [17.50, 0.406589], [17.75, 0.414627],
[18.00, 0.427837], [18.25, 0.445331], [18.50, 0.465972], [18.75, 0.488443],
[19.00, 0.511332], [19.25, 0.533222], [19.50, 0.552774], [19.75, 0.568812],
[20.00, 0.580389], [20.25, 0.586847], [20.50, 0.587849], [20.75, 0.583401],
[21.00, 0.573842], [21.25, 0.559824], [21.50, 0.542266], [21.75, 0.522293],
[22.00, 0.501167], [22.25, 0.480207], [22.50, 0.460707], [22.75, 0.443854],
CHAPTER 5. CHAPTER D

--R [19.0, 0.511332], [19.25, 0.533222], [19.5, 0.552774], [19.75, 0.568812],
--R [20.0, 0.580389], [20.25, 0.586847], [20.5, 0.587849], [20.75, 0.583401],
--R [21.0, 0.573842], [21.25, 0.559824], [21.5, 0.542266], [21.75, 0.522293],
--R [22.0, 0.501167], [22.25, 0.480207], [22.5, 0.460707], [22.75, 0.443854],
--R [23.0, 0.430662], [23.25, 0.421906], [23.5, 0.41808], [23.75, 0.419367],
--R [24.0, 0.425635], [24.25, 0.436444], [24.5, 0.451078], [24.75, 0.468594],
--R [25.0, 0.48788], [25.25, 0.507725], [25.5, 0.526986], [25.75, 0.544215],
--R [26.0, 0.558626], [26.25, 0.569272], [26.5, 0.575524], [26.75, 0.577038],
--R [27.0, 0.573666], [27.25, 0.565954], [27.5, 0.554127], [27.75, 0.539054],
--R [28.0, 0.521695], [28.25, 0.503146], [28.5, 0.485666], [28.75, 0.467104],
--R [29.0, 0.451832], [29.25, 0.439675], [29.5, 0.431359], [29.75, 0.427366],
--R [30.0, 0.427908], [30.25, 0.432913], [30.5, 0.442034], [30.75, 0.454673],
--R [31.0, 0.470019], [31.25, 0.4871], [31.5, 0.504844], [31.75, 0.522148],
--R [32.0, 0.537944], [32.25, 0.551266], [32.5, 0.561307], [32.75, 0.567471],
--R [33.0, 0.569407], [33.25, 0.567026], [33.5, 0.560508], [33.75, 0.550288],
--R [34.0, 0.537026], [34.25, 0.521566], [34.5, 0.504881], [34.75, 0.488015],
--R [35.0, 0.472012], [35.25, 0.457857], [35.5, 0.446415], [35.75, 0.438375],
--R [36.0, 0.434212], [36.25, 0.434156], [36.5, 0.438182], [36.75, 0.446014],
--R [37.0, 0.45714], [37.25, 0.470848], [37.5, 0.486272], [37.75, 0.502444],
--R [38.0, 0.518359], [38.25, 0.530331], [38.5, 0.54556], [38.75, 0.555182],
--R [39.0, 0.561321], [39.25, 0.563619], [39.5, 0.561957], [39.75, 0.556463],
--R [40.0, 0.547503], [40.25, 0.535653], [40.5, 0.521665], [40.75, 0.50642],
--R [41.0, 0.49087], [41.25, 0.47598], [41.5, 0.46267], [41.75, 0.451755],
--R [42.0, 0.443897], [42.25, 0.439656], [42.5, 0.439006], [42.75, 0.442234],
--R [43.0, 0.449025], [43.25, 0.458938], [43.5, 0.471341], [43.75, 0.485454],
--R [44.0, 0.500362], [44.25, 0.515205], [44.5, 0.529002], [44.75, 0.540923],
--R [45.0, 0.550239], [45.25, 0.556387], [45.5, 0.559004], [45.75, 0.557947],
--R [46.0, 0.553301], [46.25, 0.545374], [46.5, 0.534676], [46.75, 0.521883],
--R [47.0, 0.507802], [47.25, 0.493312], [47.5, 0.479313], [47.75, 0.466667],
--R [48.0, 0.48616], [48.25, 0.448425], [48.5, 0.44393], [48.75, 0.442936],
--R [49.0, 0.445486], [49.25, 0.451406], [49.5, 0.460311], [49.75, 0.471633],
--R [50.0, 0.484658]
--R Type: List(List(Float))
--E 2
--S 3 of 5
[[x.1, x.2, fresnelC(x.1), fresnelC(x.1) - x.2] for x in pearceyC]
--R
--R (2)
--R [[0.0, 0.0, 0.0, 0.0],
--R [0.25, 0.396461, 0.3964560954, 2000459941, -0.4579995400 59 E -8],
--R [0.5, 0.502472, 0.5024715456, 4500637208, -0.4535499362 79 E -7],
--R [0.75, 0.6531193, 0.6531193584, 292607957, 0.5842926079 57 E -7],
--R [1.0, 0.7217059, 0.7217059242 9260508777, 0.2429260508 78 E -6],
--R [1.25, 0.762404, 0.7624042531 0695862575, 0.2531069586 258 E -6],
--R [1.5, 0.779084, 0.7790837385 0396370968, -0.2614960362 903 E -6],
--R [1.75, 0.774978, 0.7749781554 8647675573, 0.1554864767 557 E -6],
--R [2.0, 0.753302, 0.7533023754 6789116559, 0.3754678911 656 E -6],
--R [2.25, 0.717446, 0.7174457114 2671496912, -0.2885732850 309 E -6],
-- R [2.5, 0.670986, 0.6709858725, 0.1274904965252 E-6],
-- R [2.75, 0.617615, 0.61761494245226, 0.5754773557398 E-7],
-- R [3.0, 0.56102, 0.5610203289781386693, 0.3289781386693 E-6],
-- R [3.25, 0.504745, 0.50474546845758580.3289781386693 E-6],
-- R [3.5, 0.452047, 0.452047147373494948325 E-6],
-- R [3.75, 0.405762, 0.4057621282011750.3289781386693 E-6],
-- R [4.0, 0.368193, 0.3681929762809747631, 0.237190252037 E-7],
-- R [4.25, 0.341021, 0.34102065443025861592, 0.34566974138406 E-6],
-- R [4.5, 0.325249, 0.3252492296755260618, 0.37513244739382 E-6],
-- R [4.75, 0.321186, 0.32118581081411496285, 0.189185885037 E-7],
-- R [5.0, 0.302645, 0.3026451155639608381, 0.247930604976 E-6],
-- R [5.25, 0.289296, 0.28929677936297990.12404040462 E-7],
-- R [5.5, 0.278922, 0.278922076806263928, 0.20614710788 E-7],
-- R [5.75, 0.268745, 0.268745147373494948325 E-6],
-- R [6.0, 0.258745, 0.258745147373494948325 E-6],
-- R [6.25, 0.248966, 0.24896677936297990.12404040462 E-7],
-- R [6.5, 0.239344, 0.23934454508057441808 E-6],
-- R [6.75, 0.229969, 0.229969211155639608381, 0.189360451357 E-7],
-- R [7.0, 0.220804, 0.220804002820237594, 0.597179767206 E-6],
-- R [7.25, 0.212186, 0.21218677936297990.12404040462 E-7],
-- R [7.5, 0.203804, 0.203804002820237594, 0.597179767206 E-6],
-- R [7.75, 0.205804, 0.205804002820237594, 0.597179767206 E-6],
-- R [8.0, 0.215087, 0.215087211155639608381, 0.189360451357 E-7],
-- R [8.25, 0.225804, 0.225804002820237594, 0.597179767206 E-6],
-- R [8.5, 0.236745, 0.236745147373494948325 E-6],
-- R [8.75, 0.248344, 0.24834454508057441808 E-6],
-- R [9.0, 0.26969, 0.269692111155639608381, 0.189360451357 E-7],
-- R [9.25, 0.290804, 0.290804002820237594, 0.597179767206 E-6],
-- R [9.5, 0.312186, 0.31218677936297990.12404040462 E-7],
-- R [9.75, 0.333804, 0.333804002820237594, 0.597179767206 E-6],
-- R [10.0, 0.355804, 0.355804002820237594, 0.597179767206 E-6],
-- R [10.25, 0.378344, 0.37834454508057441808 E-6],
-- R [10.5, 0.400804, 0.400804002820237594, 0.597179767206 E-6],
-- R [10.75, 0.423087, 0.423087211155639608381, 0.189360451357 E-7],
-- R [11.0, 0.44569, 0.445692111155639608381, 0.189360451357 E-7],
-- R [11.25, 0.469187, 0.469187211155639608381, 0.189360451357 E-7],
-- R [11.5, 0.492969, 0.492969211155639608381, 0.189360451357 E-7],
-- R [11.75, 0.517087, 0.517087211155639608381, 0.189360451357 E-7],
-- R [12.0, 0.541969, 0.541969211155639608381, 0.189360451357 E-7],
-- R [12.25, 0.566344, 0.56634454508057441808 E-6],
-- R [12.5, 0.590804, 0.590804002820237594, 0.597179767206 E-6],
-- R [12.75, 0.61469, 0.614692111155639608381, 0.189360451357 E-7],
-- R [13.0, 0.64869, 0.648692111155639608381, 0.189360451357 E-7],
-- R [13.25, 0.682644, 0.68264454508057441808 E-6],
-- R [13.5, 0.716969, 0.716969211155639608381, 0.189360451357 E-7],
-- R [13.75, 0.750483, 0.750483211155639608381, 0.189360451357 E-7],
-- R [14.0, 0.784483, 0.784483211155639608381, 0.189360451357 E-7],
-- R [14.25, 0.818187, 0.818187211155639608381, 0.189360451357 E-7],
-- R [14.5, 0.851887, 0.851887211155639608381, 0.189360451357 E-7],
-- R [14.75, 0.885682, 0.885682987070119647 E-6],
CHAPTER 5. CHAPTER D

<table>
<thead>
<tr>
<th>Frame</th>
<th>y-coordinates</th>
<th>x-coordinates</th>
<th>Width</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.0</td>
<td>0.569335</td>
<td>0.5693360588</td>
<td>834</td>
<td>202146</td>
</tr>
<tr>
<td>15.25</td>
<td>0.547984</td>
<td>0.5479846850</td>
<td>9637</td>
<td>199303</td>
</tr>
<tr>
<td>15.5</td>
<td>0.524009</td>
<td>0.5240097909</td>
<td>4969</td>
<td>992039</td>
</tr>
<tr>
<td>15.75</td>
<td>0.49893</td>
<td>0.4989302854</td>
<td>9356</td>
<td>979937</td>
</tr>
<tr>
<td>16.0</td>
<td>0.47431</td>
<td>0.4743107173</td>
<td>2032</td>
<td>792592</td>
</tr>
<tr>
<td>16.25</td>
<td>0.451659</td>
<td>0.45165966582</td>
<td>6025</td>
<td>475374</td>
</tr>
<tr>
<td>16.5</td>
<td>0.432343</td>
<td>0.4323456593</td>
<td>6678</td>
<td>17725</td>
</tr>
<tr>
<td>16.75</td>
<td>0.417502</td>
<td>0.4175027736</td>
<td>9772</td>
<td>555286</td>
</tr>
<tr>
<td>17.0</td>
<td>0.407985</td>
<td>0.4079854159</td>
<td>5598</td>
<td>154173</td>
</tr>
<tr>
<td>17.25</td>
<td>0.40433</td>
<td>0.4043020072</td>
<td>9975</td>
<td>747636</td>
</tr>
<tr>
<td>17.5</td>
<td>0.406589</td>
<td>0.4065896832</td>
<td>2726</td>
<td>31915</td>
</tr>
<tr>
<td>17.75</td>
<td>0.427837</td>
<td>0.4278371576</td>
<td>9257</td>
<td>67748</td>
</tr>
<tr>
<td>18.0</td>
<td>0.445331</td>
<td>0.4453311546</td>
<td>8967</td>
<td>96159</td>
</tr>
<tr>
<td>18.25</td>
<td>0.465972</td>
<td>0.4659716234</td>
<td>4840</td>
<td>774835</td>
</tr>
<tr>
<td>18.5</td>
<td>0.488443</td>
<td>0.4884422879</td>
<td>5458</td>
<td>921791</td>
</tr>
<tr>
<td>19.0</td>
<td>0.511332</td>
<td>0.5113318949</td>
<td>1592</td>
<td>39088</td>
</tr>
<tr>
<td>19.25</td>
<td>0.532222</td>
<td>0.5322207670</td>
<td>9031</td>
<td>239166</td>
</tr>
<tr>
<td>19.5</td>
<td>0.552774</td>
<td>0.5527754062</td>
<td>1484</td>
<td>209042</td>
</tr>
<tr>
<td>19.75</td>
<td>0.568812</td>
<td>0.5688120435</td>
<td>8656</td>
<td>883009</td>
</tr>
<tr>
<td>20.0</td>
<td>0.580389</td>
<td>0.5803889920</td>
<td>4010</td>
<td>979211</td>
</tr>
<tr>
<td>20.25</td>
<td>0.586847</td>
<td>0.5868461863</td>
<td>8595</td>
<td>115362</td>
</tr>
<tr>
<td>20.5</td>
<td>0.587849</td>
<td>0.5878413242</td>
<td>1747</td>
<td>56698</td>
</tr>
<tr>
<td>21.0</td>
<td>0.583401</td>
<td>0.5833994615</td>
<td>8142</td>
<td>05371</td>
</tr>
<tr>
<td>21.25</td>
<td>0.573842</td>
<td>0.5738402674</td>
<td>6208</td>
<td>837835</td>
</tr>
<tr>
<td>21.5</td>
<td>0.559824</td>
<td>0.5598292224</td>
<td>1791</td>
<td>771666</td>
</tr>
<tr>
<td>21.75</td>
<td>0.542266</td>
<td>0.5422647781</td>
<td>3626</td>
<td>412855</td>
</tr>
<tr>
<td>22.0</td>
<td>0.501167</td>
<td>0.5011667664</td>
<td>6519</td>
<td>922577</td>
</tr>
<tr>
<td>22.25</td>
<td>0.480207</td>
<td>0.4802071484</td>
<td>9825</td>
<td>761974</td>
</tr>
<tr>
<td>22.5</td>
<td>0.460707</td>
<td>0.4607066279</td>
<td>5313</td>
<td>545574</td>
</tr>
<tr>
<td>22.75</td>
<td>0.443654</td>
<td>0.4436547462</td>
<td>6962</td>
<td>922904</td>
</tr>
<tr>
<td>23.0</td>
<td>0.430662</td>
<td>0.4306622113</td>
<td>5307</td>
<td>45639</td>
</tr>
<tr>
<td>23.25</td>
<td>0.421906</td>
<td>0.4219061845</td>
<td>5918</td>
<td>877179</td>
</tr>
<tr>
<td>23.5</td>
<td>0.418018</td>
<td>0.4180199648</td>
<td>2406</td>
<td>114921</td>
</tr>
<tr>
<td>23.75</td>
<td>0.419367</td>
<td>0.4193672450</td>
<td>5058</td>
<td>364187</td>
</tr>
<tr>
<td>24.0</td>
<td>0.425635</td>
<td>0.4256349063</td>
<td>1204</td>
<td>612991</td>
</tr>
<tr>
<td>24.25</td>
<td>0.436444</td>
<td>0.4364434671</td>
<td>9955</td>
<td>178019</td>
</tr>
<tr>
<td>24.5</td>
<td>0.451078</td>
<td>0.4510778257</td>
<td>8097</td>
<td>250046</td>
</tr>
<tr>
<td>24.75</td>
<td>0.468594</td>
<td>0.4685940538</td>
<td>2129</td>
<td>131152</td>
</tr>
<tr>
<td>25.0</td>
<td>0.48788</td>
<td>0.487879923</td>
<td>5075</td>
<td>186185</td>
</tr>
<tr>
<td>25.25</td>
<td>0.507725</td>
<td>0.5077249997</td>
<td>4785</td>
<td>550835</td>
</tr>
<tr>
<td>25.5</td>
<td>0.526896</td>
<td>0.5268965378</td>
<td>5056</td>
<td>315834</td>
</tr>
<tr>
<td>25.75</td>
<td>0.544215</td>
<td>0.5442153996</td>
<td>5070</td>
<td>56159</td>
</tr>
<tr>
<td>26.0</td>
<td>0.558262</td>
<td>0.5582628363</td>
<td>2264</td>
<td>94663</td>
</tr>
<tr>
<td>26.25</td>
<td>0.569272</td>
<td>0.5692719365</td>
<td>6311</td>
<td>74305</td>
</tr>
<tr>
<td>26.5</td>
<td>0.575524</td>
<td>0.5755235681</td>
<td>5192</td>
<td>86467</td>
</tr>
<tr>
<td>26.75</td>
<td>0.577038</td>
<td>0.5770379859</td>
<td>3635</td>
<td>377575</td>
</tr>
<tr>
<td>27.0</td>
<td>0.573766</td>
<td>0.5737677770</td>
<td>1242</td>
<td>709811</td>
</tr>
<tr>
<td>27.25</td>
<td>0.565954</td>
<td>0.5659537051</td>
<td>6013</td>
<td>269979</td>
</tr>
</tbody>
</table>
PACKAGE DFSFUN DOUBLEFLOATSPECIALFUNCTIONS
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R

[27.5,0.554127,0.5541267540 7959236651,- 0.2459204076 335 E -6],
[27.75,0.539054,0.5390531912 4040124686,- 0.8087595987 5314 E -6],
[28.0,0.521695,0.5216949544 6705388656,- 0.4553294611 34 E -7],
[28.25,0.503146,0.5031465467 9677094812,0.5467967709 4812 E -6],
[28.5,0.484566,0.4845663019 0920749659,0.3019092074 9659 E -6],
[28.75,0.467104,0.4671043214 6483016106,0.3214648301 6106 E -6],
[29.0,0.451832,0.4518315476 4628819554,- 0.4523537118 0446 E -6],
[29.25,0.439675,0.4396743309 7300438172,- 0.6690269956 1828 E -6],
[29.5,0.431359,0.4313584765 6101829235,- 0.5234389817 0765 E -6],
[29.75,0.427366,0.4273661360 3768100434,0.1360376810 043 E -6],
[30.0,0.427908,0.4279080905 152246846,0.9051522468 46 E -7],
[30.25,0.432913,0.4329130066 7882319144,0.6678823191 44 E -8],
[30.5,0.442034,0.4420341837 397595623,0.1837397595 623 E -6],
[30.75,0.454673,0.4546732432 9642671639,0.2432964267 1639 E -6],
[31.0,0.470019,0.4700191387 880518641,0.1387880518 641 E -6],
[31.25,0.4871,0.4870999928 5423804547,- 0.7145761954 53 E -8],
[31.5,0.504844,0.5048444286 1506750598,0.4286150675 06 E -6],
[31.75,0.522148,0.5221485610 4009044596,0.5610400904 4596 E -6],
[32.0,0.537944,0.5379444615 1644558148,0.4615164455 815 E -6],
[32.25,0.551266,0.5512658647 1800830465,- 0.1352819916 953 E -6],
[32.5,0.561307,0.5613070997 1630231457,0.9971630231 457 E -7],
[32.75,0.567471,0.5674716241 0900084533,0.6241090008 4533 E -6],
[33.0,0.569407,0.5694072875 9780709976,0.2875978070 998 E -6],
[33.25,0.567026,0.5670262735 8602313224,0.2735860231 322 E -6],
[33.5,0.560508,0.5605085017 9224639203,0.5017922463 9203 E -6],
[33.75,0.550288,0.5502884623 6065133847,0.4623606513 385 E -6],
[34.0,0.537026,0.5370265183 9150336492,0.5183915033 6492 E -6],
[34.25,0.521566,0.5215664629 6901266906,0.4629690126 691 E -6],
[34.5,0.504881,0.5048817837 1619742811,0.7837161974 2811 E -6],
[34.75,0.488015,0.4880148426 3306102148,- 0.1573669389 785 E -6],
[35.0,0.472012,0.4720115763 0531557892,- 0.4236946844 2108 E -6],
[35.25,0.457857,0.4578571669 7762795948,0.1669776279 595 E -6],
[35.5,0.446415,0.4464153233 6579743027,0.3233657974 3027 E -6],
[35.75,0.438375,0.4383753574 7411594227,0.3574741159 4227 E -6],
[36.0,0.434212,0.4342121246 9792940288,0.1246979294 029 E -6],
[36.25,0.434156,0.4341557286 0605475409,- 0.2713939452 4591 E -6],
[36.5,0.438182,0.4381823268 5646395843,0.3268564639 5843 E -6],
[36.75,0.446014,0.4460136608 378042178,- 0.3391621957 8221 E -6],
[37.0,0.45714,0.4571399045 2295784095,- 0.9547704215 905 E -7],
[37.25,0.470848,0.4708481333 0533700846,0.1333053370 085 E -6],
[37.5,0.486272,0.4862712468 1703660919,- 0.7531829633 9081 E -6],
[37.75,0.502444,0.5024452759 1583009151,0.0000012759 158300915],
[38.0,0.518359,0.5183578008 5448394353,- 0.0000011991 455160565],
[38.25,0.533031,0.5330321320 7141016248,0.0000011320 714101625],
[38.5,0.54556,0.5455618898 0934649103,0.0000018898 09346491],
[38.75,0.555182,0.5551816153 550106069,- 0.3846449893 931 E -6],
[39.0,0.561321,0.5613199426 5924299224,- 0.0000010573 407570078],
[39.25,0.563619,0.5636202521 4551507916,0.0000012521 455150792],
[39.5,0.561957,0.5619620534 8342820413,0.0000050534 8342820413],
[39.75,0.556463,0.5564674768 4327383218,0.0000044768 4327383218],

339


<table>
<thead>
<tr>
<th>Value</th>
<th>Function Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>0.642119</td>
</tr>
<tr>
<td>4.25</td>
<td>0.601362</td>
</tr>
<tr>
<td>4.50</td>
<td>0.556489</td>
</tr>
<tr>
<td>4.75</td>
<td>0.510408</td>
</tr>
<tr>
<td>5.00</td>
<td>0.465942</td>
</tr>
<tr>
<td>5.25</td>
<td>0.425677</td>
</tr>
<tr>
<td>5.50</td>
<td>0.389301</td>
</tr>
<tr>
<td>5.75</td>
<td>0.356138</td>
</tr>
<tr>
<td>6.00</td>
<td>0.329852</td>
</tr>
<tr>
<td>6.25</td>
<td>0.343503</td>
</tr>
<tr>
<td>6.50</td>
<td>0.317099</td>
</tr>
<tr>
<td>6.75</td>
<td>0.290320</td>
</tr>
<tr>
<td>7.00</td>
<td>0.264286</td>
</tr>
<tr>
<td>7.25</td>
<td>0.240128</td>
</tr>
<tr>
<td>7.50</td>
<td>0.216797</td>
</tr>
<tr>
<td>7.75</td>
<td>0.194263</td>
</tr>
<tr>
<td>8.00</td>
<td>0.172439</td>
</tr>
<tr>
<td>8.25</td>
<td>0.151214</td>
</tr>
<tr>
<td>8.50</td>
<td>0.130606</td>
</tr>
<tr>
<td>8.75</td>
<td>0.110506</td>
</tr>
<tr>
<td>9.00</td>
<td>0.091007</td>
</tr>
<tr>
<td>9.25</td>
<td>0.072086</td>
</tr>
<tr>
<td>9.50</td>
<td>0.053642</td>
</tr>
<tr>
<td>9.75</td>
<td>0.035781</td>
</tr>
<tr>
<td>10.00</td>
<td>0.018499</td>
</tr>
<tr>
<td>10.25</td>
<td>0.001286</td>
</tr>
<tr>
<td>10.50</td>
<td>-0.009286</td>
</tr>
<tr>
<td>10.75</td>
<td>-0.018499</td>
</tr>
<tr>
<td>11.00</td>
<td>-0.028612</td>
</tr>
<tr>
<td>11.25</td>
<td>-0.039625</td>
</tr>
<tr>
<td>11.50</td>
<td>-0.051538</td>
</tr>
<tr>
<td>11.75</td>
<td>-0.064242</td>
</tr>
<tr>
<td>12.00</td>
<td>-0.077647</td>
</tr>
<tr>
<td>12.25</td>
<td>-0.091743</td>
</tr>
<tr>
<td>12.50</td>
<td>-0.106538</td>
</tr>
<tr>
<td>12.75</td>
<td>-0.121935</td>
</tr>
<tr>
<td>13.00</td>
<td>-0.137935</td>
</tr>
<tr>
<td>13.25</td>
<td>-0.154535</td>
</tr>
<tr>
<td>13.50</td>
<td>-0.171735</td>
</tr>
<tr>
<td>13.75</td>
<td>-0.189535</td>
</tr>
<tr>
<td>14.00</td>
<td>-0.208035</td>
</tr>
<tr>
<td>14.25</td>
<td>-0.226135</td>
</tr>
<tr>
<td>14.50</td>
<td>-0.244835</td>
</tr>
<tr>
<td>14.75</td>
<td>-0.263135</td>
</tr>
<tr>
<td>15.00</td>
<td>-0.281035</td>
</tr>
<tr>
<td>15.25</td>
<td>-0.298535</td>
</tr>
<tr>
<td>15.50</td>
<td>-0.315635</td>
</tr>
<tr>
<td>15.75</td>
<td>-0.332235</td>
</tr>
<tr>
<td>16.00</td>
<td>-0.348335</td>
</tr>
<tr>
<td>16.25</td>
<td>-0.363935</td>
</tr>
<tr>
<td>16.50</td>
<td>-0.378935</td>
</tr>
<tr>
<td>16.75</td>
<td>-0.393335</td>
</tr>
<tr>
<td>17.00</td>
<td>-0.407135</td>
</tr>
<tr>
<td>17.25</td>
<td>-0.420335</td>
</tr>
<tr>
<td>17.50</td>
<td>-0.432935</td>
</tr>
<tr>
<td>17.75</td>
<td>-0.444935</td>
</tr>
<tr>
<td>18.00</td>
<td>-0.456335</td>
</tr>
<tr>
<td>18.25</td>
<td>-0.467135</td>
</tr>
<tr>
<td>18.50</td>
<td>-0.477335</td>
</tr>
<tr>
<td>18.75</td>
<td>-0.486935</td>
</tr>
<tr>
<td>19.00</td>
<td>-0.495935</td>
</tr>
<tr>
<td>19.25</td>
<td>-0.503935</td>
</tr>
<tr>
<td>19.50</td>
<td>-0.511235</td>
</tr>
<tr>
<td>19.75</td>
<td>-0.517835</td>
</tr>
<tr>
<td>20.00</td>
<td>-0.523835</td>
</tr>
<tr>
<td>20.25</td>
<td>-0.529235</td>
</tr>
<tr>
<td>20.50</td>
<td>-0.534035</td>
</tr>
<tr>
<td>20.75</td>
<td>-0.537935</td>
</tr>
<tr>
<td>21.00</td>
<td>-0.541035</td>
</tr>
<tr>
<td>21.25</td>
<td>-0.542935</td>
</tr>
<tr>
<td>21.50</td>
<td>-0.543935</td>
</tr>
</tbody>
</table>

---

R (3)


```plaintext
--R [50.0, 0.445722]
--R Type: List(List(Float))
--E
--S 5 of 5
[[x.1, x.2, fresnelS(x.1), fresnelS(x.1) - x.2] for x in pearceyS]
```
--R [22.75, 0.561862, 0.5618616308, 0.5618616308 - 0.3691941960 79 E -6],
--R [23.0, 0.545782, 0.5457817221, 0.5457817221 - 0.2778116420 529 E -6],
--R [23.25, 0.52704, 0.5270400684, 0.5270400684 - 0.2243367771 6841 E -6],
--R [23.5, 0.486399, 0.4863982517, 0.4863982517 - 0.7482557521 2092 E -6],
--R [23.75, 0.467029, 0.4670284356, 0.4670284356 - 0.5643377462 6841 E -6],
--R [24.0, 0.449901, 0.4499008775, 0.4499008775 - 0.1224723143 032 E -6],
--R [24.25, 0.436051, 0.4360514454, 0.4360514454 - 0.445493861 0932 E -6],
--R [24.5, 0.426303, 0.4263030410, 0.4263030410 - 0.410244838 4947 E -7],
--R [25.0, 0.421217, 0.4212170480, 0.4212170480 - 0.1192396774 7 E -8],
--R [25.25, 0.421062, 0.4210613602, 0.4210613602 - 0.890881011 602 E -7],
--R [25.5, 0.425797, 0.4257967889, 0.4257967889 - 0.3307782966 303 E -6],
--R [25.75, 0.435083, 0.4350825056, 0.4350825056 - 0.4943837641 3166 E -6],
346
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R

CHAPTER 5. CHAPTER D
[35.25,0.552293,0.5522925303 0361142379,- 0.4696963885 762 E -6],
[35.5,0.540094,0.5400936428 6312004459,- 0.3571368799 554 E -6],
[35.75,0.525495,0.5254947452 6478477127,- 0.2547352152 287 E -6],
[36.0,0.509417,0.5094168513 7281277168,- 0.1486271872 283 E -6],
[36.25,0.492866,0.4928661334 9354600731,0.1334935460 073 E -6],
[36.5,0.476871,0.4768716484 1410150512,0.6484141015 0512 E -6],
[36.75,0.46242,0.4624199779 4599876237,- 0.2205400123 76 E -7],
[37.0,0.450396,0.4503960869 296054767,0.8692960547 67 E -7],
[37.25,0.441528,0.4415290525 3642844985,0.0000010525 364284499],
[37.5,0.436345,0.4363459415 0326090466,0.9415032609 0466 E -6],
[37.75,0.435144,0.4351444622 142641974,0.4622142641 974 E -6],
[38.0,0.437971,0.4379709919 0155134798,- 0.8098448652 02 E -8],
[38.25,0.444626,0.4446262686 7031444849,0.2686703144 4849 E -6],
[38.5,0.45467,0.4546708406 3699119293,0.8406369911 9293 E -6],
[38.75,0.467461,0.4674625380 8725087486,0.0000015380 872508749],
[39.0,0.482187,0.4821902969 1952370995,0.0000032969 1952370995],
[39.25,0.497924,0.4979286483 1116955857,0.0000046483 1116955857],
[39.5,0.51369,0.5136882217 6583353419,- 0.0000017782 341664658],
[39.75,0.528507,0.5285123707 6397702394,0.0000053707 6397702393],
[40.0,0.541464,0.5414672429 1598081817,0.0000032429 159808182],
[40.25,0.551768,0.5517745240 9094539537,0.0000065240 9094539537],
[40.5,0.558799,0.5588050506 5912486677,0.0000060506 5912486677],
[40.75,0.56214,0.5621369791 6412056122,- 0.0000030208 358794388],
[41.0,0.561608,0.5616247711 0814605692,0.0000167711 081460569],
[41.25,0.557258,0.5572675955 430330709,0.0000095955 430330709],
[41.5,0.549384,0.5493773200 0894645567,- 0.0000066799 9105354433],
[41.75,0.538494,0.5385616342 9485623221,0.0000676342 9485623221],
[42.0,0.525282,0.5252752111 2342898796,- 0.0000067888 7657101203],
[42.25,0.51058,0.5104939314 7702373041,- 0.0000860685 2297626959],
[42.5,0.495309,0.4953417684 038968965,0.0000327684 038968965],
[42.75,0.480418,0.4806241406 5752533599,0.0002061406 57525336],
[43.0,0.466829,0.4667676898 4497320465,- 0.0000613101 5502679535],
[43.25,0.455375,0.4553638694 6250373139,- 0.0000111305 374962686],
[43.5,0.446755,0.4466829634 3044419959,- 0.0000720365 6955580042],
[43.75,0.441487,0.4413450709 5947820527,- 0.0001419290 405217947],
[44.0,0.439878,0.4398575523 27691464,- 0.0000204476 72308536],
[44.25,0.442007,0.4420222062 7982284284,0.0000152062 798228428],
[44.5,0.44772,0.4478984397 6769945092,0.0001784397 676994509],
[44.75,0.456645,0.4559273692 7109515615,- 0.0007176307 2890484385],
[45.0,0.468209,0.4676487917 6051833044,- 0.0005602082 3948166956],
[45.25,0.481681,0.4819138773 9121176935,0.0002328773 912117694],
[45.5,0.496215,0.4954833490 8753370091,- 0.0007316509 1246629909],
[45.75,0.510904,0.5106859986 5658467469,- 0.0002180013 434153253],
[46.0,0.524837,0.5242725127 6971944478,- 0.0005644872 3028055522],
[46.25,0.537153,0.5375126919 1563903102,0.0003596919 15639031],
[46.5,0.547099,0.5460962647 9105883281,- 0.0010027352 089411672],
[46.75,0.55407,0.5524834800 5502643628,- 0.0015865199 449735637],
[47.0,0.55765,0.5580597398 2343233094,0.0004097398 234323309],
[47.25,0.557635,0.5590882082 4170920981,0.0014532082 417092098],
[47.5,0.554044,0.5475186455 874275914,- 0.0065253544 125724086],


This package provides special functions for double precision real and complex floating point.

The formula used will agree with the Table of the Fresnel Integral by Pearcey (1959) to 6 decimal places up to an argument of about 35.0. After that the summation gets slowly worse, agreeing to only 2 digits at about 45.0.

fresnelC(1.5)  
0.7790837385 0396370968

fresnelS(1.5)  
0.4154833182 6565542581

See Also:
o )show DoubleFloatSpecialFunctions
DoubleFloatSpecialFunctions (DFSFUN)

Exports:
airyAi airyBi besselI besselJ besselK
besselY Beta digamma E1 Ei
Ei1 Ei2 Ei3 Ei4 Ei5
Ei6 En fresnelC fresnelS Gamma
hypergeometric0F1 logGamma polygamma

— package DFSFUN DoubleFloatSpecialFunctions —

)abbrev package DFSFUN DoubleFloatSpecialFunctions
++ Author: Bruce W. Char, Timothy Daly, Stephen M. Watt
++ Date Created: 1990
++ Date Last Updated: Jan 19, 2008
++ Description:
++ This package provides special functions for double precision
++ real and complex floating point.

DoubleFloatSpecialFunctions(): Exports == Impl where
NNI ==> NonNegativeInteger
PI ==> Integer
R ==> DoubleFloat
C ==> Complex DoubleFloat
OPR ==> OnePointCompletion R
F ==> Float
LF ==> List Float

Exports ==> with
Gamma: R -> R
++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
++ \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.  
Gamma: C -> C
++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
++ \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.  
E1: R -> OPR
++ E1(x) is the Exponential Integral function
++ The current implementation is a piecewise approximation
++ involving one poly from -4..4 and a second poly for x > 4

En: (PI,R) -> OPR
++ En(n,x) is the nth Exponential Integral Function

Ei: (OPR) -> OPR
++ Ei is the Exponential Integral function
++ This is computed using a 6 part piecewise approximation.
++ DoubleFloat can only preserve about 16 digits but the
++ Chebyshev approximation used can give 30 digits.

Ei1: (OPR) -> OPR
++ Ei1 is the first approximation of Ei where the result is
++ x*%e^-x*Ei(x) from -infinity to -10 (preserves digits)

Ei2: (OPR) -> OPR
++ Ei2 is the first approximation of Ei where the result is
++ x*%e^-x*Ei(x) from -10 to -4 (preserves digits)

Ei3: (OPR) -> OPR
++ Ei3 is the first approximation of Ei where the result is
++ (Ei(x)-log |x| - gamma)/x from -4 to 4 (preserves digits)

Ei4: (OPR) -> OPR
++ Ei4 is the first approximation of Ei where the result is
++ x*%e^-x*Ei(x) from 4 to 12 (preserves digits)

Ei5: (OPR) -> OPR
++ Ei5 is the first approximation of Ei where the result is
++ x*%e^-x*Ei(x) from 12 to 32 (preserves digits)

Ei6: (OPR) -> OPR
++ Ei6 is the first approximation of Ei where the result is
++ x*%e^-x*Ei(x) from 32 to infinity (preserves digits)

Beta: (R, R) -> R
++ Beta(x, y) is the Euler beta function, \spad{\text{\text{\text{\text{B(x,y)}}}}}, defined by
++ \spad{\text{\text{\text{\text{integrate(t^{(x-1)}(1-t)^{(y-1)}, t=0..1)}}}}}. 
++ This is related to \spad{\text{\text{\text{\text{Gamma(x)}}}}} by
++ \spad{\text{\text{\text{\text{integrate(t^{(x-1)}(1-t)^{(y-1)}, t=0..1)}}}}}. 

Beta: (C, C) -> C
++ Beta(x, y) is the Euler beta function, \spad{\text{\text{\text{\text{B(x,y)}}}}}, defined by
++ \spad{\text{\text{\text{\text{integrate(t^{(x-1)}(1-t)^{(y-1)}, t=0..1)}}}}}. 
++ This is related to \spad{\text{\text{\text{\text{Gamma(x)}}}}} by
++ \spad{\text{\text{\text{\text{integrate(t^{(x-1)}(1-t)^{(y-1)}, t=0..1)}}}}}. 

logGamma: R -> R
++ logGamma(x) is the natural log of \spad{\text{\text{\text{\text{Gamma(x)}}}}}.
logGamma: C -> C
++ logGamma(x) is the natural log of \spad{Gamma(x)}.
++ This can often be computed even if \spad{Gamma(x)} cannot.

digamma: R -> R
++ digamma(x) is the function, \spad{psi(x)}, defined by
++ \spad{psi(x) = Gamma'(x)/Gamma(x)}.

digamma: C -> C
++ digamma(x) is the function, \spad{psi(x)}, defined by
++ \spad{psi(x) = Gamma'(x)/Gamma(x)}.

polygamma: (NNI, R) -> R
++ polygamma(n, x) is the n-th derivative of \spad{digamma(x)}.

besselJ: (R, R) -> R
++ besselJ(v, x) is the Bessel function of the first kind,
++ \spad{J(v, x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.

besselJ: (C, C) -> C
++ besselJ(v, x) is the Bessel function of the first kind,
++ \spad{J(v, x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.

besselY: (R, R) -> R
++ besselY(v, x) is the Bessel function of the second kind,
++ \spad{Y(v, x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \spad{Y(v, x) = (J(v, x) cos(v*%pi) - J(-v, x))/sin(v*%pi)}
++ so is not valid for integer values of v.

besselY: (C, C) -> C
++ besselY(v, x) is the Bessel function of the second kind,
++ \spad{Y(v, x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \spad{Y(v, x) = (J(v, x) cos(v*%pi) - J(-v, x))/sin(v*%pi)}
++ so is not valid for integer values of v.

besselI: (R, R) -> R
++ besselI(v, x) is the modified Bessel function of the first kind,
++ \spad{I(v, x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
besselI: (C,C) -> C
++ besselI(v,x) is the modified Bessel function of the first kind,
++ \ \spad{I(v,x)}.
++ This function satisfies the differential equation:
++ \ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.

besselK: (R, R) -> R
++ besselK(v,x) is the modified Bessel function of the second kind,
++ \ \spad{K(v,x)}.
++ This function satisfies the differential equation:
++ \ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \ \spad{K(v,x) = \pi/2*(I(-v,x) - I(v,x))/\sin(v*\pi)}.
++ so is not valid for integer values of v.

besselK: (C, C) -> C
++ besselK(v,x) is the modified Bessel function of the second kind,
++ \ \spad{K(v,x)}.
++ This function satisfies the differential equation:
++ \ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \ \spad{K(v,x) = \pi/2*(I(-v,x) - I(v,x))/\sin(v*\pi)}
++ so is not valid for integer values of v.

airyAi: C -> C
++ airyAi(x) is the Airy function \spad{Ai(x)}.
++ This function satisfies the differential equation:
++ \ \spad{Ai''(x) - x \cdot Ai(x) = 0}.

airyAi: R -> R
++ airyAi(x) is the Airy function \spad{Ai(x)}.
++ This function satisfies the differential equation:
++ \ \spad{Ai''(x) - x \cdot Ai(x) = 0}.

airyBi: R -> R
++ airyBi(x) is the Airy function \spad{Bi(x)}.
++ This function satisfies the differential equation:
++ \ \spad{Bi''(x) - x \cdot Bi(x) = 0}.

airyBi: C -> C
++ airyBi(x) is the Airy function \spad{Bi(x)}.
++ This function satisfies the differential equation:
++ \ \spad{Bi''(x) - x \cdot Bi(x) = 0}.

hypergeometric0F1: (R, R) -> R
++ hypergeometric0F1(c,z) is the hypergeometric function
++ \ \spad{0F1(c;z)}.

hypergeometric0F1: (C, C) -> C
++ hypergeometric0F1(c,z) is the hypergeometric function
++ \ \spad{0F1(c;z)}.

def: F -> F
++ def(f) denotes the Fresnel integral S
The Exponential Integral

The E1 function

(Quoted from Segletes[7]):

A number of useful integrals exist for which no exact solutions have been found. In other cases, an exact solution, if found, may be impractical to utilize over the complete domain of the function because of precision limitations associated with what usually ends up as a series solution to the challenging integral. For many of these integrals, tabulated values may be published in various mathematical handbooks and articles. In some handbooks, fits (usually piecewise) also are offered. In some cases, an application may be forced to resort to numerical integration in order to acquire the integrated function. In this context, compact (i.e. not piecewise) analytical fits to some of these problematic integrals, accurate to within a small fraction of the numerically integrated value, serve as a useful tool to applications requiring the results of the integration, especially when the integration is required numerous times throughout the course of the application. Furthermore, the ability and methodology to develop intelligent fits, in contract to the more traditional “brute force” fits, provide the means to minimize parameters and maximize accuracy when tackling some of these difficult functions. The exponential integral will be used as an opportunity to both demonstrate a methodology for intelligent fitting as well as for providing an accurate, compact, analytical fit to the exponential integral.

The exponential integral is a useful class of functions that arise in a variety of applications [...]. The real branch of the family of exponential integrals may be defined as

\[ E_n(x) = x^{n-1} \int_x^\infty \frac{e^{-t}}{t^n} \, dt \] (5.1)

where \( n \), a positive integer, denotes the specific member of the exponential integral family.
The argument of the exponential integral, rather than expressing a lower limit of integration as in (1), may be thought of as describing the exponential decay constant, as given in this equivalent (and perhaps more popular) definition of the integral:

\[ E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} \, dt \]  

(5.2)

Integration by parts permits any member of the exponential integral family to be converted to an adjacent member of the family, by way of

\[ \int_x^\infty \frac{e^{-t}}{t^{n+1}} \, dt = \frac{1}{n} \left( \frac{e^{-x}}{x^n} - \int_x^\infty \frac{e^{-t}}{t^n} \, dt \right) \]  

(5.3)

expressable in terms of \( E_n \) as

\[ E_{n+1}(x) = \frac{1}{n} \left[ e^{-x} - xE_n(x) \right] \quad (n = 1, 2, 3) \]  

(5.4)

Through recursive employment of this equation, all members of the exponential integral family may be analytically related. However, this technique only allows for the transformation of one integral into another. There remains the problem of evaluating \( E_1(x) \). There is an exact solution to the integral of \( (e^{-t}/t) \), appearing in a number of mathematical references [?, ?] which is obtainable by expanding the exponential into a power series and integrating term by term. That exact solution, which is convergent, may be used to specify \( E_1(x) \) as

\[ E_1(x) = -\gamma - \ln(x) + \frac{x}{1!} - \frac{x^2}{2!} + \frac{x^3}{3!} - \ldots \]  

(5.5)

Euler’s constant, \( \gamma \), equal to 0.57721..., arises when the power series expansion for \( (e^{-t}/t) \) is integrated and evaluated at its upper limit, as \( x \to \infty \). [?]

Employing eqn (5), however, to evaluate \( E_1(x) \) is problematic for finite \( x \) significantly larger than unity. One may well ask of the need to evaluate the exponential integral for large \( x \), since the function to be integrated drops off so rapidly that the integral is surely a very flat function. Such reasoning is true when comparing the integrand at large \( x \) to that at small \( x \). However, the definition of eqn (1) has as its upper limit not a small value of \( x \), but rather that of \( \infty \). Therefore, the actual values for \( E_n(x) \) are extremely small numbers for large values of \( x \). This is not sufficient merely to select enough terms of eqn (5) to evaluate the integral to within a value of, for example \( \pm 0.0001 \) because the actual integral value for large \( x \) would be smaller than this arbitrary tolerance. To draw an analogy, it would be like saying that it is good enough to approximate \( e^{-x} \) as 0.0 for \( x > 10 \), since its actual value is within 0.0001 of zero. For some applications, such an approximation may be warranted. In general, though, such an approximation is mathematically unacceptable. Worse yet, as seen from eqns (1) and (2), the need to evaluate the exponential integral for large arguments can arise in real-world problems from either a large integration limit or a large value of an exponential decay constant. Thus, the need to evaluate exponential integrals for large values of the argument is established. It is here that the practical problems with the evaluation of eqn (5) become manifest.
First, the number of terms, \( N \), required to achieve convergence rises rapidly with increasing \( x \), making the summation an inefficient tool, even when expressed as a recursion relation (for three digits of accuracy, \( N \) is observed to vary roughly as \( 9 + 1.6x \), for \( 1 < x < 7 \)). More important, however, is the fact that, for calculations of finite precision, the accuracy of the complete summation will be governed by the individual term of greatest magnitude. The source of the problem is that as \( x \) is increased, the total summation decreases in magnitude more rapidly than a decaying exponential, while at the same time, the largest individual term in the series is observed to grow rapidly with increasing \( x \) (\( \sim 10^1 \) for \( x = 7 \), \( \sim 10^2 \) for \( x = 10 \), \( \sim 10^3 \) for \( x = 13 \), etc.). The magnitude of this largest individual term consumes the available precision and, as a result, leaves little or none left for the ever-diminishing net sum that constitutes the desired integral.

Literally, the use of eqn (5), even with (32-bit) double precision, does not permit the exponential integral to be evaluated to three places for \( x > 14 \) in any case, and with the situation worsening for lesser precision. For these reasons, the use of eqn (5) to evaluate the exponential integral numerically for large \( x \) is wholly unsuitable.

\[
E_1(x) = e^{-x} \cdot \frac{1}{x + \frac{1}{x + \frac{2}{x + \ldots}}} \tag{5.6}
\]

But as \( x \) becomes smaller, the number of terms required for convergence rises quickly. Similar arguments apply for the use of an asymptotic expansion for \( E_1 \), which also converges for large \( x \). As such, the more typical approach employed by handbooks is that of a fit. While some steps are taken to make the fits intelligent (e.g., transformation of variables), the fits are all piecewise over the domain of the integral.

Cody and Thatcher [?] performed what is perhaps the definitive work, with the use of Chebyshev[? , ?] approximations to the exponential integral \( E_1 \). Like others, they fit the integral over a piecewise series of subdomains (three in their case) and provide the fitting parameters necessary to evaluate the function to various required precisions, down to relative errors of \( 10^{-20} \). One of the problems with piecewise fitting over two or more subdomains is that functional value and derivatives of the spliced fits will not, in general, match at the domain transition point, unless special accommodations are made. This sort of discontinuity in functional value and/or slope, curvature, etc., may cause difficulties for some numerical algorithms operating upon the fitted function. Numerical splicing/smoothing algorithms aimed at eliminating discontinuities in the value and/or derivatives of a piecewise fit are not, in general, computationally insignificant. Problems associated with piecewise splicing of fits may also be obviated by obtaining an accurate enough fit, such that the error is on the order of magnitude of the limiting machine precision. This alternative, however, requires the use of additional fitting parameters to acquire the improved precision. Thus, regardless of approach, the desire to eliminate discontinuities in the function and its derivatives, between piecewise splices, requires extra computational effort. One final benefit to be had by avoiding the use of piecewise fits is the concomitant avoidance of conditional (i.e., IF...THEN) programming.
statements in the coding of the routine. The use of conditional statements can preclude maximum computing efficiency on certain parallel computing architectures.

Segletes constructs an analytic, non-piecewise fit to the Exponential Integral but the precision is on the order of 4 decimal places and is not sufficient to compare against the Abramowitz and Stegun Handbook.

Instead we have chosen to use a two piece fitting function based on the Chebyshev polynomial for computing $E_1$. This agrees with the handbook values to almost the last published digit. See the e1.input pamphlet for regression testing against the handbook tables.

**E1**: $\mathbb{R} \rightarrow \mathbb{O} \mathbb{P} \mathbb{R}$

The special function E1 below was originally derived from a function written by T. Haavie as the `expint.c` function in the Numlibc library by Lars Erik Lund. Haavie approximates the E1 function by two Chebyshev polynomials. For the range $-4 < x < 4$ the Chebyshev coefficients are:

$$\begin{align*}
7.8737715392882774, & -8.0314874286705335, & 3.8797325768522250, \\
-1.6042971072992259, & 0.5630905453891458, & -0.1704423017433357, \\
0.045209939015415, & -0.0106538986439085, & 0.0022562638123478, \\
-0.004335700473221, & 0.0000762166811878, & -0.0000123417443064, \\
0.000018519745698, & -0.0000002588698662, & 0.000000338604319, \\
-0.000000041611418, & 0.000000004821606, & -0.00000000528465, \\
0.00000000054945, & -0.00000000005433, & 0.0000000000512, \\
-0.00000000000046, & 0.00000000000004
\end{align*}$$

and for the range $x > 4$ the Chebyshev coefficients are:

$$\begin{align*}
0.2155283776715125, & 0.1028106215227030, & -0.0045526707131788, \\
0.000357163122851, & -0.000379341616932, & 0.00004914394914, \\
-0.0000073565024922, & 0.0000001230603606, & -0.0000000225236907, \\
0.00000004412375, & -0.000000009328509, & 0.000000002069297, \\
-0.0000000000481502, & 0.000000000116891, & -0.000000000029474, \\
0.000000000007691, & -0.000000000002070, & 0.00000000000583, \\
-0.0000000000000163, & 0.0000000000000047, & -0.0000000000000014, \\
0.0000000000000004, & -0.0000000000000001
\end{align*}$$

I've rewritten the polynomial to use precomputed coefficients that take into account the scaling used by Haavie. I've also rewritten the polynomial using Horner's method so the large powers of $x$ are only computed once.

The result can be either a double float or, if the argument is zero, infinity. Thus we need to extend the result to be a one-point completion to include infinity.

---

| package DFSFUN DoubleFloatSpecialFunctions |

E1(x:R):OPR ==

x = 0.0:.R => infinity()
x > 4.0::R =>
t1:R:=0.14999948967737774608E-15::R
t2:R:=0.999999999993112::R
ta:R:=(t1*x+t2)
t3:R:=0.99999999953685760001::R
tb:R:=(ta*x-t3)
t4:R:=1.999998808293376::R
tc:R:=(tb*x+t4)
t5:R:=5.999983407661056::R
td:R:=(tc*x-t5)
t6:R:=23.998538093848166::R
te:R:=(td*x+t6)
t7:R:=119.9108830382784512::R
tf:R:=(te-x-t7)
t8:R:=716.0135102092017661::R
tg:R:=(tf*x+t8)
t9:R:=4903.346662337098573::R
th:R:=(tg*x-t9)
t10:R:=36601.2584145446674::R
ti:R:=(th*x+t10)
t11:R:=279913.28608482691646::R
tj:R:=(ti-x-t11)
t12:R:=2060518.7020296525186::R
tk:R:=(tj*x+t12)
t13:R:=13859772.093039815059::R
tl:R:=(tk*x-t13)
t14:R:=81945572.630072918857::R
tm:R:=(tl*x+t14)
t15:R:=413965714.82128317479::R
tn:R:=(tm*x-t15)
t16:R:=1747209536.2595547568::R
to:R:=(tn*x+t16)
t17:R:=6036121233.96179427::R
tp:R:=(to*x-t17)
t18:R:=16693883576.106267572::R
tq:R:=(tp*x+t18)
t19:R:=3593862564.582862097::R
tr:R:=(tq*x-t19)
t20:R:=57888657293.609258888::R
ts:R:=(tr*x+t20)
t21:R:=65523779423.11290127::R
tt:R:=(ts*x-t21)
t22:R:=46422751473.201760309::R
tu:R:=(tt*x+t22)
t23:R:=15474250491.067253436::R
tv:R:=(tu*x-t23)
tw:R:=(-1.0::R*x)
tx:R:=exp(tw)
ty:R:=tv*tx
tz:R:=x**22
taz := \frac{ty}{tz}

x > -4.0::R =>

a1 := 0.476837158203125E-22::R
a2 := 0.10967254638671875E-20::R
a3 := 0.20217895507812500001E-19::R
a4 := 0.42600631713867187501E-18::R
a5 := 0.86862564086914062501E-17::R
a6 := 0.16553192138671875E-15::R
a7 := 0.29870208740234375E-14::R
a8 := 0.5097890777587890625E-13::R
a9 := 0.81934069213867499E-12::R
ah := 0.1235313123779296875E-10::R
a10 := 0.1739729620849609375E-9::R
a11 := 0.22774642697021484375E-8::R
a12 := 0.275573192853515625E-7::R
a13 := 0.30619243635087890625E-6::R
a14 := 0.1235313123779296875E-5::R
a15 := 0.00003100198412519140625E+0::R
a16 := 0.00002834467120045546875E+0::R
a17 := 0.000023148148148176953125E+0::R
a18 := 0.00166666666666646875E+0::R
a19 := 0.01041666666666646875E+0::R
a20 := 0.05555555555555554168751E+0::R
a21 := 0.25000000000000000375E+0::R
a22 := 1.00000000000000325E+0::R
a23 := 0.5772156649015328::R
a24 := -1.0::R*\log(\text{abs}(x)) + av
aw := 0::FR
The $E_n$ function is computed using the recurrence relation:

$$E_{n+1}(z) = \frac{1}{n} \left( e^{-z} - zE_n(z) \right) \quad (n = 1, 2, 3, \ldots)$$

The base case of the recursion depends on $E_1$ above.

The formula is 5.1.14 in Abramowitz and Stegun, 1965, p229[?].

— package DFSFUN DoubleFloatSpecialFunctions —

```lisp
En(n:PI,x:R):OPR ==
  n=1 => E1(x)
  v:R:=retract(En((n-1)::PI,x))
  w:R:=1/(n-1)*(exp(-x)-x*v)
  w::OPR
```

The Ei Function

This function is based on Kin L. Lee’s work[?]. See also [?].

Abstract

The exponential integral $Ei(x)$ is evaluated via Chebyshev series expansion of its associated functions to achieve high relative accuracy throughout the entire real line. The Chebyshev coefficients for these functions are given to 30 significant digits. Clenshaw’s[?] method is modified to furnish an efficient procedure for the accurate solution of linear systems having near-triangular coefficient matrices.

Introduction

The evaluation of the exponential integral

$$Ei(x) = \int_{-\infty}^{x} \frac{e^u}{u} \, du = -E_1(-x), x \neq 0$$

is usually based on the value of its associated functions, for example, $xe^{-x}Ei(x)$. High accuracy tabulations of integral (1) by means of Taylor series techniques are given by Harris
[?] and Miller and Hurst [?]. The evaluation of \( Ei(x) \) for \(-4 \leq x \leq \infty\) by means of Chebyshev series is provided by Clenshaw [?] to have the absolute accuracy of 20 decimal places. The evaluation of the same integral (1) by rational approximation of its associated functions is furnished by Cody and Thacher [?, ?] for \(-\infty < x < \infty\), and has the relative accuracy of 17 significant figures.

The approximation of Cody and Thacher from the point of view of efficient function evaluation are preferable to those of Clenshaw. However, the accuracy of the latter’s procedure, unlike those of the former, is not limited by the accuracy or the availability of a master function, which is a means of explicitly evaluating the function in question.

In this paper \( Ei(x) \) (or equivalently \(-E_1(-x)\)) for the entire real line is evaluated via Chebyshev series expansion of its associated functions that are accurate to 30 significant figures by a modification of Clenshaw’s procedure. To verify the accuracy of the several Chebyshev series, values of the associated functions were checked against those computed by Taylor series and those of Murnaghan and Wrench [?] (see Remarks on Convergence and Accuracy).

Although for most purposes fewer than 30 figures of accuracy are required, such high accuracy is desirable for the following reasons. In order to further reduce the number of arithmetical operations in the evaluation of a function, the Chebyshev series in question can either be converted into a rational function or rearranged into an ordinary polynomial. Since several figures may be lost in either of these procedures, it is necessary to provide the Chebyshev series with a sufficient number of figures to achieve the desired accuracy. Furthermore, general function approximation routines, such as those used for minimax rational function approximations, require the explicit evaluation of the function to be approximated. To take account of the errors committed by these routines, the function values must have an accuracy higher than the approximation to be determined. Consequently, high-precision results are useful as a master function for finding approximations for (or involving) \( Ei(x) \) (e.g. [?, ?]) where prescribed accuracy is less than 30 figures.

**Discussion**

It is proposed here to provide for the evaluation of \( Ei(x) \) by obtaining Chebyshev coefficients for the associated functions given by table 1.

**Table 1:** Associated Functions of \( Ei(x) \) and their ranges of Chebyshev Series Expansions

<table>
<thead>
<tr>
<th>Associated function</th>
<th>Range of expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Ei_1 ) ( xe^{-x}Ei(x) )</td>
<td>(-\infty &lt; x \leq -10)</td>
</tr>
<tr>
<td>( Ei_2 ) ( xe^{-x}Ei(x) )</td>
<td>(-10 \leq x \leq -4)</td>
</tr>
<tr>
<td>( Ei_3 ) ( \frac{Ei(x) - \log</td>
<td>x</td>
</tr>
<tr>
<td>( Ei_4 ) ( xe^{-x}Ei(x) )</td>
<td>(4 \leq x \leq 12)</td>
</tr>
<tr>
<td>( Ei_5 ) ( xe^{-x}Ei(x) )</td>
<td>(12 \leq x \leq 32)</td>
</tr>
<tr>
<td>( Ei_6 ) ( xe^{-x}Ei(x) )</td>
<td>(32 \leq x &lt; \infty)</td>
</tr>
</tbody>
</table>

(\( \gamma = 0.5772156649... \) is Euler’s constant.)
Note that the functions \( Ei(x) = x - \log|\log x|/x \) and \( x e^{-x} Ei(x) \) have the limiting values of unity at the origin and at infinity, respectively, and that the range of the associated function values is close to unity (see table 4). This makes for the evaluation of the associated functions over the indicated ranges in table 1 (and thus \( Ei(x) \) over the entire real line) with high relative accuracy by means of the Chebyshev series. The reason for this will become apparent later.

Some remarks about the choice of the intervals of expansion for the several Chebyshev series are in order here. The partition of the real line indicated by table 1 is chosen to allow for the approximation of the associated functions with a maximum error of \( 0.5 \times 10^{-30} \) by polynomials of degree \( < 50 \). The real line has also been partitioned with the objective of providing the interval about zero with the lowest degree of polynomial approximation of the six intervals. This should compensate for the computation of \( \log|x| \) required in the evaluation of \( Ei(x) \) over that interval. The ranges \(-\infty < x \leq -4 \) and \( 4 \leq x < \infty \) are partitioned into 2 and 3 intervals, respectively, to provide approximations to \( x e^{-x} Ei(x) \) by polynomials of about the same degree.

Expansions in Chebyshev Series

Let \( \phi(t) \) be a differentiable function defined on \([-1,1]\). To facilitate discussion, denote its Chebyshev series and that of its derivative by

\[
\phi(t) = \sum_{k=0}^{\infty} a_k^{(0)} T_k(t) \quad \phi'(t) = \sum_{k=0}^{\infty} a_k^{(1)} T_k(t)
\]
where \( T_k(t) \) are Chebyshev polynomials defined by
\[
T_k(t) = \cos(k \arccos t), \quad -1 \leq t \leq 1
\]
(A prime over a summation sign indicates that the first term is to be halved.)

If \( \phi(t) \) and \( \phi'(t) \) are continuous, the Chebyshev coefficients \( A_k^{(0)} \) and \( A_k^{(1)} \) can be obtained analytically (if possible) or by numerical quadrature. However, since each function in table 1 satisfies a linear differential equation with polynomial coefficients, the Chebyshev coefficients can be more readily evaluated by the method of Clenshaw [?].

There are several variations of Clenshaw’s procedure (see, e.g. [?]), but for high-precision computation, where multiple precision arithmetic is employed, we find his original procedure easiest to implement. However, straightforward application of it may result in a loss of accuracy if the trial solutions selected are not sufficiently independent. How the difficulty is overcome will be pointed out subsequently.

The function \( xe^{-x}\text{Ei}(x) \) on the Finite Interval

We consider first the Chebyshev series expansion of
\[
f(x) = xe^{-x}\text{Ei}(x), \quad (a \leq x \leq b)
\]
where \( x \neq 0 \). One can easily verify that after the change of variables
\[
x = [(b - a)T + a + b]/2, \quad (-1 \leq t \leq 1)
\]
the function
\[
\phi(t) = f \left[ \frac{(b - a)t + a + b}{2} \right] = f(x)
\]
satisfies the differential equation
\[
2(pt + q)\phi'(t) + p(pt + q - 2)\phi(t) = p(pt + q)
\]
with
\[
\phi(-1) = ae^{-a}\text{Ei}(a)
\]
where \( p = b - a \) and \( q = b + a \). Replacing \( \phi(t) \) and \( \phi'(t) \) in equations 7 by their Chebyshev series, we obtain
\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]
\[
2 \sum_{k=0}^{\infty} A_k^{(1)}(pt + q)T_k(t) + p \sum_{k=0}^{\infty} A_k^{(0)}(pt + q - 2)T_k(t) = p(pt + q)
\]
\[
\text{The value of } \text{Ei}(a) \text{ may be evaluated by means of the Taylor series. In this report } \text{Ei}(a) \text{ is computed by first finding the Chebyshev series approximation to } [\text{Ei}(x) - \log|x| - \gamma]/x \text{ to get } \text{Ei}(a). \text{ The quantities } e^a \text{ and } \log|a| \text{ for integral values of } a \text{ may be found in existing tables.}
It can be demonstrated that if \( B_k \) are the Chebyshev coefficients of a function \( \Psi(t) \), then \( C_k \), the Chebyshev coefficients of \( t^r \Psi(t) \) for positive integers \( r \), are given by

\[
C_k = 2^{-r} \sum_{i=0}^{r} \binom{r}{i} B_{|k-r+2i|} \quad (5.17)
\]

Consequently, the left member of equation 15 can be rearranged into a single series involving \( T_k(t) \). The comparison of the coefficients of \( T_k(t) \) that yields the infinite system of equations

\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]

\[
2pA_{k-1}^{(1)} + 4qA_k^{(1)} + 2pA_{k+1}^{(1)} + p^2A_{k-1}^{(0)} + 2p(q-2)A_k^{(0)} + p^2A_{k+1}^{(0)} = \begin{cases} 
4pq & k = 0 \\
2p^2 & k = 1 \\
0 & k = 2, 3, \ldots
\end{cases} \quad (5.18)
\]

The relation \( 2kA_k^{(0)} = A_{k-1}^{(1)} - A_{k+1}^{(1)} \) \( (5.19) \) can be used to reduce equation 18 to a system of equations involving only \( A_k^{(0)} \). Thus, replacing \( k \) of equations 18 by \( k + 2 \) and subtracting the resulting equation from equations 18, we have, by means of equation 19, the system of equations

\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]

\[
2p(q-2)A_0 + (8q + p^2)A_1 + 2p(6-q)A_2 - p^2A_3 = 4pq
\]

\[
p^2A_{k-1} + 2p(2k + q - 2)A_k + 8q(k+1)A_{k+1} + 2p(2k - q + 6)A_{k+2} - p^2A_{k+3} = \begin{cases} 
2p^2 & k = 1 \\
0 & k = 2, 3, \ldots
\end{cases} \quad (5.20)
\]

The superscript of \( A_k^{(0)} \) is dropped for simplicity. In order to solve the infinite system 20, Clenshaw \( [?] \) essentially considered the required solution as the limiting solution of the sequence of truncated systems consisting of the first \( M + 1 \) equations of the same system, that is, the solution of the system

\[
\sum_{k=0}^{M} (-1)^k A_k = \phi(-1)
\]

\[
2p(q-2)A_0 + (8q + p^2)A_1 + 2p(q - 6)A_2 - p^2A_3 = 4pq \quad (5.21)
\]
\[ \begin{align*}
    p^2 A_{k-1} + 2p(2k + q - 2)A_k + 8q(k + 1)A_{k+1} + 2p(2k - q + 6)A_{k+2} - p^2 A_{k+3} \\
    = \begin{cases} 
        2p^2, & k = 1 \\
        0, & k = 2, 3, \ldots, M - 3
    \end{cases} \\
    p^2 A_{M-3} + 2p(2M + q - 6)A_{M-2} + 8q(M - 1)A_{M-1} + 2p(2M + 4 - q)A_M = 0 \\
    p^2 A_{M-2} + 2p(2M + q - 4)A_{M-1} + 8qMA_M = 0
\end{align*} \] (5.23)

where \( A_k \) is assumed to vanish for \( K \geq M + 1 \). To solve system (21,22,23) consider first the subsystem 23 consisting of \( M - 2 \) equations in \( M \) unknowns. Here use is made of the fact that the subsystem 23 is satisfied by

\[ A_k = c_1 \alpha_k + c_2 \beta_k + \gamma_k \quad (k = 0, 1, 2, \ldots) \] (5.24)

for arbitrary constants \( c_1 \) and \( c_2 \), where \( \gamma_k \) is a particular solution of 23 and where \( \alpha_k \) and \( \beta_k \) are two independent solutions of the homogeneous equations (23 with \( 2p^2 \) deleted) of the same subsystem. Hence, if \( \alpha_k \), \( \beta_k \), and \( \gamma_k \) are available, the solution of system (21,22,23) reduces to the determinant of \( c_1 \) and \( c_2 \) from equations 21 and 22.

To solve equations (21,22,23), we note that

\[ \gamma_0 = 2, \quad \gamma_k = 0, \quad \text{for } k = 1(1)M \] (5.25)

is obviously a particular solution of equation 23. The two independent solutions \( \gamma_k \) and \( \beta_k \) of the homogeneous equations of the same subsystem can be generated in turn by backward recurrence if we set

\[ \begin{align*}
    &\alpha_{M-1} = 0, \quad \alpha_M = 1 \\
    &\beta_{M-1} = 1, \quad \beta_M = 0
\end{align*} \] (5.26)

or choose any \( \alpha_M - 1, \alpha_M, \) and \( \beta_M - 1, \beta_M \) for which \( \alpha_{M-1} \beta_M - \alpha_M \beta_{M-1} \neq 0 \). The arbitrary constants \( c_1 \) and \( c_2 \) are determined, and consequently the solution of equations (21,22,23) if equation 24 is substituted into equation 21 and 22 and the resulting equations

\[ c_1 R(\alpha) + c_2 R(\beta) = \phi(-1) - 1 \] (5.27)

\[ c_1 S(\alpha) + c_2 S(\beta) = 8p \] (5.28)

are solved as two equations in two unknowns. The terms \( R(\alpha) \) and \( S(\alpha) \) are equal, respectively, to the left members of equations 21 and 22 corresponding to solution \( \alpha_k \). (The identical designation holds for \( R(\beta) \) and \( S(\beta) \).)

The quantities \( \alpha_k \) and \( \beta_k \) are known as trial solutions in reference [?]. Clenshaw has pointed out that if \( \alpha_k \) and \( \beta_k \) are not sufficiently independent, loss of significance will occur in the formation of the linear combination 24, with consequent loss of accuracy. Clenshaw suggested the Gauss-Seidel iteration procedure to improve the accuracy of the solution. However, this requires the application of an additional computing procedure and may prove to be extremely slow. A simpler procedure which does not alter the basic computing scheme given above is
proposed here. The loss of accuracy can effectively be regained if we first generate a third trial solution \( \delta_k \) (\( k=0,1, \ldots, M \)), where \( \delta_{M-1} \) and \( \delta_M \) are equal to \( c_1 \alpha_{M-1} + c_2 \beta_{M-1} \) and \( c_1 \alpha_M + c_2 \beta_M \), respectively, and where \( \delta_k \) (\( k=M-2, M-3, \ldots, 0 \)) is determined using backward recurrence as before by means of equation 23. Then either \( \alpha_k \) or \( \beta_k \) is replaced by \( \delta_k \) and a new set of \( c_1 \) and \( c_2 \) is determined by equations 27 and 28. Such a procedure can be repeated until the required accuracy is reached. However, only one application of it was necessary in the computation of the coefficients of this report.

As an example, consider the case for \( 4 \leq x \leq 12 \) with \( M = 15 \). The right member of equation 27 and of equation 28 assume, respectively, the values of 0.43820800 and 64. The trial solutions \( \alpha_k \) and \( \beta_k \) generated with \( \alpha_{14} = 8, \alpha_{15} = 9 \) and \( \beta_{14} = 7, \beta_{15} = 8 \) are certainly independent, since \( \alpha_{14} \beta_{15} - \alpha_{15} \beta_{14} = 1 \neq 0 \). A check of table 2 shows that equations 27 and 28 have, respectively, the residuals of \(-0.137 \times 10^{-4} \) and \(-0.976 \times 10^{-3} \). The same table also shows that \( c_1 \alpha_k \) is opposite in sign but nearly equal in magnitude to \( c_2 \beta_k \). Cancellations in the formation of the linear combination 24 causes a loss of significance of 2 to 6 figures in the computed \( A_k \). In the second iteration, where a new set of \( \beta_k \) is generated replacing \( \beta_{14} \) and \( \beta_{15} \), respectively, by \( c_1 \alpha_{14} + c_2 \beta_{14} \) and \( c_1 \alpha_{15} + c_2 \beta_{15} \) of the first iteration, the new \( c_1 \alpha_k \) and \( c_2 \beta_k \) differed from 2 to 5 orders of magnitude. Consequently, no cancellation of significant figures in the computation of \( A_k \) occurred. Notice that equations 27 and 28 are now satisfied exactly. Further note that the new \( c_1 \) and \( c_2 \) are near zero and unity, respectively, for the reason that if equations 21, 22, and 23 are satisfied by equation 24 exactly in the first iteration, the new \( c_1 \) and \( c_2 \) should have the precise values zero and 1, respectively. The results of the third iteration show that the \( A_k \) of the second iteration are already accurate to eight decimal places, since the \( A_k \) in the two iterations differ in less that \( 0.5 \times 10^{-8} \). Notice that for the third iteration, equations 27 and 28 are also satisfied exactly and that \( c_1 = 1 \) and \( c_2 = 0 \) (relative to 8 places of accuracy).

**Table 2**: Computation of Chebyshev Coefficients for \( xe^{-x} Ei(x) \)

| First iteration: | \( \alpha_{14} = 8 \), \( \alpha_{15} = 9 \); \( \beta_{14} = 7 \), \( \beta_{15} = 8 \) |
### Table 1: Values of $c_1 \alpha_k$, $c_2 \beta_k$, and $A_k$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_1 \alpha_k$</th>
<th>$c_2 \beta_k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.71690285E 03</td>
<td>-0.71644773E 03</td>
<td>0.24551200E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.33302683E 03</td>
<td>0.33284640E 03</td>
<td>-0.16243000E 00</td>
</tr>
<tr>
<td>2</td>
<td>0.13469341E 03</td>
<td>-0.13464845E 03</td>
<td>0.44960000E 01</td>
</tr>
<tr>
<td>3</td>
<td>-0.43211869E 02</td>
<td>0.43205127E 02</td>
<td>-0.67420000E 02</td>
</tr>
<tr>
<td>4</td>
<td>0.99929173E 01</td>
<td>-0.99942238E 01</td>
<td>-0.13065000E 02</td>
</tr>
<tr>
<td>5</td>
<td>-0.11670764E 01</td>
<td>0.11684754E 01</td>
<td>0.13810000E 02</td>
</tr>
<tr>
<td>6</td>
<td>-0.25552137E 00</td>
<td>0.25493635E 00</td>
<td>-0.58502000E 02</td>
</tr>
<tr>
<td>7</td>
<td>0.20617247E 00</td>
<td>-0.20599754E 00</td>
<td>0.17493000E 03</td>
</tr>
<tr>
<td>8</td>
<td>-0.75797238E 01</td>
<td>0.75756775E 01</td>
<td>-0.40471000E 04</td>
</tr>
<tr>
<td>9</td>
<td>0.20550680E 01</td>
<td>-0.20543463E 01</td>
<td>0.37950000E 07</td>
</tr>
<tr>
<td>10</td>
<td>-0.45192333E 02</td>
<td>0.45183721E 02</td>
<td>-0.86120000E 06</td>
</tr>
<tr>
<td>11</td>
<td>0.82656562E 03</td>
<td>-0.82656589E 03</td>
<td>0.27000000E 09</td>
</tr>
<tr>
<td>12</td>
<td>-0.12335751E 03</td>
<td>0.12337366E 03</td>
<td>0.39500000E 07</td>
</tr>
<tr>
<td>13</td>
<td>0.13300910E 04</td>
<td>-0.13315328E 04</td>
<td>-0.14418000E 07</td>
</tr>
<tr>
<td>14</td>
<td>-0.29699001E 06</td>
<td>0.30091136E 06</td>
<td>0.39813500E 08</td>
</tr>
<tr>
<td>15</td>
<td>-0.33941716E 06</td>
<td>0.33852528E 06</td>
<td>-0.89188000E 09</td>
</tr>
</tbody>
</table>

\[c_1 = 0.37613920E - 07\]
\[c_2 = -0.42427144E - 07\]
\[c_1 R(\alpha) + c_2 R(\beta) - 0.43820800E 00 = -0.13700000E - 04\]
\[c_1 S(\alpha) + c_2 S(\beta) - 0.64000000E 00 = -0.97600000E - 03\]

Second iteration: $\alpha_{14} = 8, \alpha_{15} = 9$;
$\beta_{14} = 0.39213500E - 08, \beta_{15} = -0.89188000E - 09$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_1 \alpha_k$</th>
<th>$c_2 \beta_k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.36701576E -05</td>
<td>0.45512986E 00</td>
<td>0.24551355E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.17051695E -05</td>
<td>-0.16243837E 00</td>
<td>-0.16243000E 00</td>
</tr>
<tr>
<td>2</td>
<td>0.68976566E -06</td>
<td>0.44956834E 01</td>
<td>0.49957523E 01</td>
</tr>
<tr>
<td>3</td>
<td>-0.22132756E -06</td>
<td>-0.67413538E -02</td>
<td>-0.67415751E -02</td>
</tr>
<tr>
<td>4</td>
<td>0.51197561E -07</td>
<td>-0.13067496E -02</td>
<td>-0.13066984E -02</td>
</tr>
<tr>
<td>5</td>
<td>-0.59856744E -08</td>
<td>0.13810895E 02</td>
<td>0.13810835E 02</td>
</tr>
<tr>
<td>6</td>
<td>-0.13059663E -08</td>
<td>-0.58502164E -03</td>
<td>-0.58502294E -03</td>
</tr>
<tr>
<td>7</td>
<td>0.10552667E -08</td>
<td>0.17492889E -03</td>
<td>0.17492994E -03</td>
</tr>
<tr>
<td>8</td>
<td>-0.38808033E -09</td>
<td>-0.40472426E -04</td>
<td>-0.40472814E -04</td>
</tr>
<tr>
<td>9</td>
<td>0.10523831E -09</td>
<td>0.72169965E 05</td>
<td>0.72171017E 05</td>
</tr>
<tr>
<td>10</td>
<td>-0.23146333E -10</td>
<td>-0.86125438E -06</td>
<td>-0.86127752E -06</td>
</tr>
<tr>
<td>11</td>
<td>0.42342615E -11</td>
<td>-0.25542252E -09</td>
<td>-0.25118825E -09</td>
</tr>
<tr>
<td>12</td>
<td>-0.63200810E -12</td>
<td>0.37946968E -07</td>
<td>0.37946336E -07</td>
</tr>
<tr>
<td>13</td>
<td>0.68210630E -13</td>
<td>-0.14417584E -07</td>
<td>-0.14417516E -07</td>
</tr>
<tr>
<td>14</td>
<td>-0.15414832E -14</td>
<td>0.39212981E -08</td>
<td>0.39212965E -08</td>
</tr>
<tr>
<td>15</td>
<td>-0.17341686E -14</td>
<td>-0.89186818E -09</td>
<td>-0.89186991E -09</td>
</tr>
</tbody>
</table>

\[c_1 = -0.19268540E - 15\]
\[c_2 = 0.99998675E 00\]
\[c_1 R(\alpha) + c_2 R(\beta) - 0.43820800E 00 = 0.0\]
\[ c_1 S(\alpha) + c_2 S(\beta) - 0.64000000 E\] 00 = 0.0

Table 2: Computation of Chebyshev Coefficients for \( x e^{-x} E_i(x) \) - Concluded
\[ 4 \leq x \leq 12 \text{ with } M = 15; \gamma_0 = 2, \gamma_k = 0 \text{ for } k = 1(1)15 \]

Third iteration: \( \alpha_{14} = 8, \alpha_{15} = 9; \beta_{14} = 0.39212965 E - 08, \beta_{15} = -0.89186991 E - 09 \)

<table>
<thead>
<tr>
<th>k</th>
<th>( c_1 \alpha_k )</th>
<th>( c_2 \beta_k )</th>
<th>( A_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.23083059E-07</td>
<td>0.45513355E 00</td>
<td>0.24551335E 01</td>
</tr>
<tr>
<td>1</td>
<td>0.10724479E-07</td>
<td>-0.16243838E 00</td>
<td>-0.16243837E 00</td>
</tr>
<tr>
<td>2</td>
<td>-0.43382065E-08</td>
<td>0.44957526E-01</td>
<td>0.44957522E-01</td>
</tr>
<tr>
<td>3</td>
<td>0.13920157E-08</td>
<td>-0.67415759E-02</td>
<td>-0.67415745E-02</td>
</tr>
<tr>
<td>4</td>
<td>-0.3220152E-09</td>
<td>0.13810835E-02</td>
<td>0.13810836E-02</td>
</tr>
<tr>
<td>5</td>
<td>0.37646251E-10</td>
<td>-0.58502297E-03</td>
<td>-0.58502296E-03</td>
</tr>
<tr>
<td>6</td>
<td>0.82137336E-11</td>
<td>-0.58502297E-03</td>
<td>-0.58502296E-03</td>
</tr>
<tr>
<td>7</td>
<td>-0.66369857E-11</td>
<td>0.17492995E-03</td>
<td>0.17492994E-03</td>
</tr>
<tr>
<td>8</td>
<td>0.24407892E-11</td>
<td>-0.40472817E-04</td>
<td>-0.40472814E-04</td>
</tr>
<tr>
<td>9</td>
<td>-0.66188494E-12</td>
<td>0.72171023E-05</td>
<td>0.72171017E-05</td>
</tr>
<tr>
<td>10</td>
<td>0.14557636E-12</td>
<td>-0.86127766E-06</td>
<td>-0.86127751E-06</td>
</tr>
<tr>
<td>11</td>
<td>-0.26630930E-13</td>
<td>0.25116620E-09</td>
<td>0.25116623E-09</td>
</tr>
<tr>
<td>12</td>
<td>0.39749465E-14</td>
<td>-0.13066983E-02</td>
<td>-0.13066986E-02</td>
</tr>
<tr>
<td>13</td>
<td>-0.42900337E-15</td>
<td>0.13810835E-02</td>
<td>0.13810836E-02</td>
</tr>
<tr>
<td>14</td>
<td>0.96949915E-17</td>
<td>-0.86127766E-06</td>
<td>-0.86127751E-06</td>
</tr>
<tr>
<td>15</td>
<td>0.10906865E-16</td>
<td>-0.89186992E-09</td>
<td>-0.89186990E-09</td>
</tr>
</tbody>
</table>

\[ c_1 = 0.12118739 E - 17 \]
\[ c_2 = 0.10000000 E 01 \]
\[ c_1 R(\alpha) + c_2 R(\beta) - 0.43820800 E\] 00 = 0.0
\[ c_1 S(\alpha) + c_2 S(\beta) - 0.64000000 E\] 00 = 0.0

It is worth noting that the coefficient matrix of system (21, 22, 23) yields an upper triangular matrix of order \( M - 1 \) after the deletion of the first two rows and the last two columns. Consequently, the procedure of this section is applicable to any linear system having this property. As a matter of fact, the same procedure can be generalized to solve linear systems having coefficient matrices of order \( N \), the deletion of whose first \( r \) \((r < N)\) rows and last \( r \) columns yields upper triangular matrices of order \( N - r \).

The Function \((1/x)[Ei(x) - \log|x| - \gamma]\)

Let
\[ f(x) = (1/x)[Ei(x) - \log|x| - \gamma], \; g(x) = e^x, \; |x| \leq b \] \hspace{1cm} (5.29)

These functions, with the change of variable \( x = bt \), simultaneously satisfy the differential equations
\[ bt^2 \phi'(t) + bt \phi(t) - \psi(t) = -1 \] \hspace{1cm} (5.30)
\[ \psi'(t) - b \psi(t) = 0, \; -1 \leq t \leq 1 \] \hspace{1cm} (5.31)
Conversely,\(^2\) any solution of equations 30 and 31 is equal to the functions given by equations 29 for the change of variable \(x = bt\). Therefore, boundary conditions need not be imposed for the solution of the differential equations.

A procedure similar to that of the previous section gives the coupled infinite recurrence relations

\[
\begin{align*}
&bA_1 + bA_3 - B_0 + B_2 = -2 \\
&kbA_{k-1} + 2(k + 1)bA_{k+1} + (k + 2)bA_{k+3} - 2B_k + 2B_{k+2} = 0 \\
&bB_{k-1} - 2kB_k - bB_{k+1} = 0, \quad k = 1, 2, \ldots
\end{align*}
\]

(5.32)

(5.33)

where \(A_k\) and \(B_k\) are the Chebyshev coefficients of \(\phi(t)\) and \(\psi(t)\), respectively.

Consider first the subsystem 33. If \(A_k = \alpha_k\) and \(B_k = \beta_k\) are a simultaneous solution of the system, which is homogeneous, then

\[
\begin{align*}
&A_k = c\alpha_k \\
&B_k = c\beta_k
\end{align*}
\]

(5.34)

are also a solution for an arbitrary constant \(c\). Thus based on considerations analogous to the solution of equations 21, 22, and 23, one can initiate an approximate solution of equations 32 and 33 by setting

\[
\begin{align*}
&\alpha_M = 0, \quad \alpha_k = 0 \quad \text{for } k \geq M + 1 \\
&\beta_M = 1, \quad \beta_k = 0 \quad \text{for } k \geq M + 1
\end{align*}
\]

(5.35)

and then determining \(\alpha_k\) and \(\beta_k\) \((k = M - 1, M - 2, \ldots, 0)\) by backward recurrence by means of equation 33. The arbitrary constant \(c\) is determined by substituting 34 into 32.

The Function \(xe^{-x}Ei(x)\) on the Infinite Interval

Let

\[
f(x) = xe^{-x}Ei(x), \quad -\infty < x \leq b < 0, \quad \text{or} 0 < b \leq x < \infty
\]

(5.36)

By making the change of variables,

\[
x = \frac{2b}{t+1}
\]

(5.37)

we can easily demonstrate that

\[
f(x) = f\left[\frac{2b}{t+1}\right] = \phi(t)
\]

(5.38)

\(^2\)The general solution of the differential equations has the form

\[
\phi(t) = (c_1/t) + [Ei(bt) - \log|bt| - \gamma]/bt
\]

\[
\psi(t) = c_2e^{bt}
\]

where the first and second terms of \(\phi(t)\) are, respectively, the complementary solution and a particular integral of equation 30. The requirement that \(\phi(t)\) is bounded makes the constant \(c_1 = 0\). The fact that \(\psi(0) = 1\) is implicit in equation 30.
satisfies the differential equation
\[(t + 1)^2 \phi'(t) + (t + 1 - 2b)\phi(t) = -2b\] (5.39)

with
\[\phi(1) = be^{-b}Ei(b)\] (5.40)

An infinite system of equations involving the Chebyshev coefficients \(A_k\) of \(\phi(t)\) is deducible from equations 39 and 40 by the same procedure as applied to equations 13 and 14 to obtain the infinite system 20; it is given as follows.

\[\sum_{k=0}^{\infty} A_k = \phi(1) = be^{-b}Ei(b)\] (5.41)

\[(1 - 2b)A_0 + 3A_1 + (3 + 2b)A_2 + A_3 = -4b\] (5.42)

\[kA_{k-1} + 2[(2k + 1) - 2b]A_k + 6(k + 1)A_{k+1} + 2(2k + 3 + 2b)A_{k+2} + (k + 2)A_{k+3} = 0, \quad k = 1, 2, \ldots\] (5.43)

As in the case of equations 21, 22 and 23, the solution of 41, 42 and 43 can be assumed to be
\[A_k = c_1\alpha_k + c_2\beta_k\] (5.44)

with \(A_k\) vanishing for a \(k \geq M\). Thus, we can set, say

\[\begin{align*}
\alpha_{M-1} &= 0, & \alpha_M &= 1 \\
\beta_{M-1} &= 1, & \beta_M &= 0
\end{align*}\] (5.45)

and determine the trial solutions \(\alpha_k\) and \(\beta_k\) (\(k=M-1,M-2,\ldots,0\)) by means of equation 43 by backward recurrence. The required solution of equations 41, 42, and 43 is then determined by substituting equation 44 in equations 41 and 42 and solving the resulting equations for \(c_1\) and \(c_2\).

Loss of accuracy in the computation of \(A_k\) can also occur here, as in the solution of equations 21, 22 and 23, if the trial solutions are not sufficiently independent. The process used to improve the accuracy of \(A_k\) of the system 21, 22 and 23 can also be applied here.

For efficiency in computation, it is worth noting that for \(b < 0\) (\(-\infty < x \leq b < 0\)) the boundary condition 40 is not required for the solution of equation 39 and 40. This follows from the fact that any solution of the differential equation 39 is equal to \(xe^{-x}Ei(x)\) \((x = 2b/(t + 1))\). Hence the \(A_k\) of \(xe^{-x}Ei(x)\) for \(-\infty < x \leq b < 0\) can be obtained without the use of equation 39 and can be assumed to have the form
\[A_k = c\alpha_k, \quad (k = 0, 1, \ldots, M)\] (5.46)

The \(M+1\) values of \(\alpha_k\) can be generated by setting \(\alpha_M = 1\) and computing \(\alpha_k\) (\(k=0,1,\ldots,M-1\)) by means of equation 43 by backward recurrence. The substitution of equation 46 into 42 then enables one to determine \(c\) from the resulting equation.

---

3The general solution of the differential equation 39. Since equation 39 has no bounded complementary solution for \(-\infty < x \leq b < 0\), every solution of it is equal to the particular integral \(xe^{-x}Ei(x)\). On the other hand, a solution of equation 39 for \(0 < x \leq b < \infty\) would, in general, involve the complementary function. Hence, boundary condition 40 is required to guarantee that the solution of equation 39 is equal to \(xe^{-x}Ei(x)\).
Remarks on Convergence and Accuracy

The Chebyshev coefficients of table 3 were computed on the IBM 7094 with 50-digit normalized floating-point arithmetic. In order to assure that the sequence of approximate solutions (see Discussion) converged to the limiting solution of the differential equation in question, a trial $M$ was incremented by 4 until the approximate Chebyshev coefficients showed no change greater than or equal to $0.5 \times 10^{-35}$. Hence the maximum error is bounded by

$$0.5(M + 1) \times 10^{-35} + \sum_{M+1}^{\infty} |A_k|$$

where the first term is the maximum error of the $M+1$ approximate Chebyshev coefficients, and the sum is the maximum error of the truncated Chebyshev series of $M+1$ terms. If the Chebyshev series is rapidly convergent, the maximum error of the approximate Chebyshev series should be of the order of $10^{-30}$. The coefficients of table 3 have been rounded to 30 digits, and higher terms for $k > N$ giving the maximum residual

$$\sum_{k=N+1}^{M} |A_k| < 0.5 \times 10^{-30}$$

have been dropped. This should allow for evaluation of the relevant function that is accurate to 30 decimal places. Since the range of values of each function is bounded between $2/5$ and 5, the evaluated function should be good to 30 significant digits. Taylor series evaluation also checks with that of the function values of table 4 (computed with 30-digit floating-point arithmetic using the coefficients of table 3) for at least $28-1/2$ significant digits. Evaluation of $\text{Ei}(x)$ using the coefficients of table 3 also checked with Murnaghan and Wrench [?] for $28-1/2$ significant figures.

Table 3: Chebyshev Coefficients (a)

$$xe^{-x} \text{Ei}(x) = \sum_{k=0}^{40} tA_k T_k(t), \quad t = (-20/x) - 1, \quad (-\infty < x \leq -10)$$
CHAPTER 5. CHAPTER D

<table>
<thead>
<tr>
<th>k</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1912173225 8605534539 1519326510E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.4208355052 868483755 0974986860E 01</td>
</tr>
<tr>
<td>2</td>
<td>0.1722819627 284362683 7118157835E 02</td>
</tr>
<tr>
<td>3</td>
<td>-0.9915782173 4445636455 984232973E 04</td>
</tr>
<tr>
<td>4</td>
<td>0.176093168 0227750526 5590665592E 05</td>
</tr>
<tr>
<td>5</td>
<td>-0.6152733145 095216982 896231293E 07</td>
</tr>
<tr>
<td>6</td>
<td>0.6024857106 5627583129 590665592E 07</td>
</tr>
<tr>
<td>7</td>
<td>-0.9915782173 4445636455 984232973E 04</td>
</tr>
<tr>
<td>8</td>
<td>0.1722819627 284362683 7118157835E 02</td>
</tr>
<tr>
<td>9</td>
<td>-0.4208355052 868483755 0974986860E 01</td>
</tr>
<tr>
<td>10</td>
<td>0.1397704123 226768027 823486823E 10</td>
</tr>
<tr>
<td>11</td>
<td>-0.2051008376 7838095818 490798351E 12</td>
</tr>
<tr>
<td>12</td>
<td>0.3168388726 0024778181 490798351E 12</td>
</tr>
<tr>
<td>13</td>
<td>-0.5132760082 891806541 598471899E 13</td>
</tr>
<tr>
<td>14</td>
<td>0.6806933040 7665493418 743368738E 14</td>
</tr>
<tr>
<td>15</td>
<td>-0.1527015040 930849719 875235531E 14</td>
</tr>
<tr>
<td>16</td>
<td>0.2784688261 549375965 0105251453E 15</td>
</tr>
<tr>
<td>17</td>
<td>-0.5249890437 421766080 847293696E 16</td>
</tr>
<tr>
<td>18</td>
<td>0.1020717991 248516294 7455787226E 16</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>package DFSFUN DoubleFloatSpecialFunctions</th>
</tr>
</thead>
</table>

$E_{i1}(y:\text{OFR}) \equiv$

finite? y => 1
x:R:=retract(y)
t:R:=acos((-20.0::R/x)-1.0::R)::R

| $t_{01}$ | 0.1912173225 8605534539 1519326510E 01 |
| $t_{02}$ | 0.1397704123 226768027 823486823E 10 |
| $t_{03}$ | 0.1722819627 284362683 7118157835E 02 |
| $t_{04}$ | 0.3168388726 0024778181 490798351E 12 |
| $t_{05}$ | 0.6806933040 7665493418 743368738E 14 |
| $t_{06}$ | 0.2784688261 549375965 0105251453E 15 |
| $t_{07}$ | 0.5249890437 421766080 847293696E 16 |
| $t_{08}$ | 0.1020717991 248516294 7455787226E 16 |

---

0.366271481 032002590 1177078922E-30
Table 3: Chebyshev Coefficients - Continued (b)

\[ xe^{-x} Ei(x) = \sum_{k=0}^{40} A_k T_k(t), \quad t = (x + 7)/3, \quad (-10 \leq x \leq -4) \]
CHAPTER 5. CHAPTER D

---

<table>
<thead>
<tr>
<th>( k )</th>
<th>( A_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1757554694 0612937384 8762834691E+01</td>
</tr>
<tr>
<td>1</td>
<td>-0.4358541517 7361661170 5001862628 034691E+01</td>
</tr>
<tr>
<td>2</td>
<td>-0.7979507139 5584254013 3217027492E+02</td>
</tr>
<tr>
<td>3</td>
<td>-0.1484372327 3071121385 970210010E+02</td>
</tr>
<tr>
<td>4</td>
<td>-0.2800301984 3775147518 620934948E+03</td>
</tr>
<tr>
<td>5</td>
<td>-0.5384614812 8657932303 9177361553E+04</td>
</tr>
<tr>
<td>6</td>
<td>-0.1032867243 5735486611 023266640E+04</td>
</tr>
<tr>
<td>7</td>
<td>-0.2041083313 0553687773 226194865E+05</td>
</tr>
<tr>
<td>8</td>
<td>-0.3961758434 2786875529 8765762816E+21</td>
</tr>
<tr>
<td>9</td>
<td>-0.7853872767 0966136306 7607560967E-07</td>
</tr>
<tr>
<td>10</td>
<td>-0.1567925981 0074698262 4616270279E-07</td>
</tr>
<tr>
<td>11</td>
<td>-0.3150055939 3763998825 007372851E-08</td>
</tr>
<tr>
<td>12</td>
<td>-0.6365096822 5242037340 830623972E-09</td>
</tr>
<tr>
<td>13</td>
<td>-0.1292888113 2805631835 6593121259E-09</td>
</tr>
<tr>
<td>14</td>
<td>-0.2638690999 6573548661 023266640E-10</td>
</tr>
<tr>
<td>15</td>
<td>-0.5408958287 4506873419 192207896E-11</td>
</tr>
<tr>
<td>16</td>
<td>-0.1080622764 9066136306 7607560967E-11</td>
</tr>
<tr>
<td>17</td>
<td>-0.2209642726 0443642618 433864145E-12</td>
</tr>
<tr>
<td>18</td>
<td>-0.4766632839 4951902622 391342091E-13</td>
</tr>
<tr>
<td>19</td>
<td>-0.9911756747 3352709450 850263972E-14</td>
</tr>
<tr>
<td>20</td>
<td>-0.2067103580 4957072400 805005201E-14</td>
</tr>
</tbody>
</table>

---

\[
Ei2(y:OPR):OPR == \\
x:=\text{retract}(y) \\
t:=\cos((x+7.0::R)/3.0::R)::R \\
t01:= 0.175755649606129373848762834691E1::R*cos(0.0::R)/2.0::R \\
t02:=t01-0.435854151773616611705001867964E-01::R*cos(t) \\
t03:=t02-0.7979507139558425401323210727492E-02::R*cos( 2.0::R*t) \\
t04:=t03-0.1484372327307112138590972010001E-02::R*cos( 3.0::R*t) \\
t05:=t04-0.280030198437751457486203954948E-03::R*cos( 4.0::R*t) \\
t06:=t05-0.538461481286579323039177361553E-04::R*cos( 5.0::R*t) \\
t07:=t06-0.10328672436735548661023266460E-05::R*cos( 6.0::R*t) \\
t08:=t07-0.20410833130565368773226198633E-05::R*cos( 7.0::R*t) \\
t09:=t08-0.396175843427386645822338443500E-06::R*cos( 8.0::R*t) \\
t10:=t09-0.785387276709663163067607656069E-07::R*cos( 9.0::R*t) \\
t11:=t10-0.156792598100746982624616270279E-07::R*cos(10.0::R*t) \\
t12:=t11-0.315005593937639988250007372851E-08::R*cos(11.0::R*t) \\
t13:=t12-0.636509682252420373040380263972E-09::R*cos(12.0::R*t) \\
t14:=t13-0.12928881132805651385659121259E-09::R*cos(13.0::R*t) \\
t15:=t14-0.263869099965926576132149942808E-10::R*cos(14.0::R*t) \\
t16:=t15-0.5408958287045086734912207896E-11::R*cos(15.0::R*t) \\
t17:=t16-0.1113222784601089999767692708E-11::R*cos(16.0::R*t) \\
t18:=t17-0.2299624726074462461843386415E-12::R*cos(17.0::R*t) \\
t19:=t18-0.476668238951902623931482091E-13::R*cos(18.0::R*t)
Table 3: Chebyshev Coefficients - Continued (c)

\[
\frac{[Ei - \log|x| - \gamma]}{x} = \sum_{k=0}^{33} A_k T_k(t), \quad t = x/4, \quad (-4 \leq x \leq 4)
\]
\[
\begin{array}{c|c|c}
k & A_k & \text{coefficient} \\
\hline
0 & 0.329370103 & 7673912939 3905231421E 01 \\
1 & 0.167835052 & 3713029156 5505796064E 01 \\
2 & 0.7220436105 & 678543524 0299697644E 00 \\
3 & 0.2600312360 & 5480956171 3740181192E 00 \\
4 & 0.8010494308 & 1737502239 4742889237E-01 \\
5 & 0.2151403663 & 9763337548 0552483005E-01 \\
6 & 0.5116207789 & 187503312602 1968984121E-02 \\
7 & 0.109032861 & 073913560 5066199014E-02 \\
8 & 0.210475320 & 2393891631 8348675226E-03 \\
9 & 0.370004516 & 6518885709 5940815956E-04 \\
10 & 0.6043491637 & 238787570 4767032866E-05 \\
11 & 0.905264273 & 0626005264 9596541772E-06 \\
12 & 0.1273805160 & 659247886 5567184969E-06 \\
13 & 0.166915748 & 4109890739 0981431841E-07 \\
14 & 0.2054417026 & 4010479254 7612484551E-08 \\
15 & 0.2383584444 & 4668176591 4052321417E-09 \\
16 & 0.2615386378 & 8854429666 9068664148E-10 \\
17 & 0.272158622 & 8541670644 6550268995E-11 \\
18 & 0.293750031 & 9835792992 5326427442E-12 \\
19 & 0.2412224096 & 7072635546 7884089307E-13 \\
20 & 0.2290130406 & 8650370941 8510620516E-14 \\
21 & 0.197546573 & 0746229940 1057650412E-15 \\
22 & 0.1634024551 & 9289317406 8635419984E-16 \\
23 & 0.129835437 & 0796376099 196219204E-17 \\
24 & 0.0992587925 & 0737105964 4632581302E-19 \\
25 & 0.0700625006 & 7221032944 723088087E-20 \\
26 & 0.0518967683 & 404451272 078008019E-21 \\
27 & 0.036049454 & 0997068112 8043162222E-22 \\
28 & 0.0236167943 & 2793864237 187203946E-23 \\
29 & 0.0151637767 & 2145092754 6924516189E-24 \\
30 & 0.0093908722 & 2448744292 5310405245E-26 \\
31 & 0.0056722755 & 9903921198 9581737831E-27 \\
32 & 0.0033833627 & 7954330315 6597939562E-28 \\
33 & 0.0019006260 & 0081619452 6680482373E-29 \\
\end{array}
\]

\[\gamma = 0.5772156649 015328606 6512090082 E 00\]

---

**package DFSFUN DoubleFloatSpecialFunctions**

\[
\text{Ei3}(y:OFR) := \begin{align*}
x := \text{retract}(y) \\
t := \text{acos}(x/4.0::R) \\
t01 := 0.32937010376739129399305231421E1::R*cos(0.0::R)/2.0::R \\
t02 := t01 + 0.16783505237130291565505796064E1::R*cos(t) \\
t03 := t02 + 0.722043610567854352402996794440E0::R*cos(2.0::R*t) \\
t04 := t03 + 0.260301236504809561713740181992E0::R*cos(3.0::R*t) \\
t05 := t04 + 0.80049403817375023294742889237E-01::R*cos(4.0::R*t) \\
t06 := t05 + 0.215140366397633375480552483005E-01::R*cos(5.0::R*t) \\
t07 := t06 + 0.51162077893033120621968910894E-02::R*cos(6.0::R*t) \\
t08 := t07 + 0.10903286100739135605066190014E-02::R*cos(7.0::R*t) \\
t09 := t08 + 0.210741532023938916318348675226E-03::R*cos(8.0::R*t) \\
t10 := t09 + 0.3279045166518885709594815956E-04::R*cos(9.0::R*t) \\
t11 := t10 + 0.60439163712387875704767032866E-05::R*cos(10.0::R*t) \\
t12 := t11 + 0.909296427936260952649596541772E-06::R*cos(11.0::R*t) \\
t13 := t12 + 0.127380516065926478865561849696E-06::R*cos(12.0::R*t) \\
t14 := t13 + 0.1669157484109890739096143814E-07::R*cos(13.0::R*t) \\
t15 := t14 + 0.20544170264010479254761248551E-08::R*cos(14.0::R*t) \\
t16 := t15 + 0.23835844444681765914052324147E-09::R*cos(15.0::R*t) \\
t17 := t16 + 0.261538637888544296669068664414E-10::R*cos(16.0::R*t) \\
t18 := t17 + 0.272158862285416706446550268995E-11::R*cos(17.0::R*t) \\
t19 := t18 + 0.269375003198357929925326427442E-12::R*cos(18.0::R*t) \\
t20 := t19 + 0.254122094670726355467884089307E-13::R*cos(19.0::R*t) \\
t21 := t20 + 0.229013040686503709418510620516E-14::R*cos(20.0::R*t)
\end{align*}
\]
t22 := t21 + 0.1975465739074299401057650412E-15 :: R * cos(21.0 :: R * t)
t23 := t22 + 0.1634024551928931740686635419984E-16 :: R * cos(22.0 :: R * t)
t24 := t23 + 0.129823643707963760991961293204E-17 :: R * cos(23.0 :: R * t)
t25 := t24 + 0.992258792507371059644632581302E-19 :: R * cos(24.0 :: R * t)
t26 := t25 + 0.7306252806722103294473230880087E-20 :: R * cos(25.0 :: R * t)
t27 := t26 + 0.518967683460434512720780080019E-21 :: R * cos(26.0 :: R * t)
t28 := t27 + 0.356040945409970681128043162227E-22 :: R * cos(27.0 :: R * t)
t29 := t28 + 0.236197943257938642370187203948E-23 :: R * cos(28.0 :: R * t)
t30 := t29 + 0.151683776772145297549624516819E-24 :: R * cos(29.0 :: R * t)
t31 := t30 + 0.94390897224487442925310405245E-26 :: R * cos(30.0 :: R * t)
t32 := t31 + 0.569722755950369211989581737831E-27 :: R * cos(31.0 :: R * t)
t33 := t32 + 0.33383362779543303156597939562E-28 :: R * cos(32.0 :: R * t)
t34 := t33 + 0.190062601281619148526680482237E-29 :: R * cos(33.0 :: R * t)
t34 :: OPR

Table 3: Chebyshev Coefficients - Continued (d)

\[ xe^{-x}Ei(x) = \sum_{k=0}^{49} A_k T_k(t), \quad t = (x - 8)/4, \quad (4 \leq x \leq 12) \]
\[
\begin{array}{cc}
k & A_k \\
0 & 0.2455133538 \\
1 & -0.1624383791 \\
2 & 0.4495753080 \\
3 & -0.6741578679 \\
4 & -0.1306697142 \\
5 & 0.1381083146 \\
6 & -0.5850228790 \\
7 & 0.1749299341 \\
8 & -0.4047281499 \\
9 & 0.7217102412 \\
10 & -0.8612776970 \\
11 & 0.2019219891 \\
12 & -0.4612950261 \\
13 & 0.3794747138 \\
14 & -0.1442117969 \\
15 & 0.3935049295 \\
16 & -0.9284689401 \\
17 & 0.2031789568 \\
18 & -0.4292498504 \\
19 & 0.8992647177 \\
20 & -0.2514475296 \\
21 & 0.3794747138 \\
22 & -0.8612776970 \\
23 & 0.2019219891 \\
24 & -0.4612950261 \\
25 & 0.3935049295 \\
26 & -0.9284689401 \\
27 & 0.2031789568 \\
28 & -0.4292498504 \\
29 & 0.8992647177 \\
30 & -0.2514475296 \\
31 & 0.3794747138 \\
32 & -0.8612776970 \\
33 & 0.2031789568 \\
34 & -0.4292498504 \\
35 & 0.8992647177 \\
36 & -0.2514475296 \\
37 & 0.3794747138 \\
38 & -0.8612776970 \\
39 & 0.2031789568 \\
40 & -0.4292498504 \\
41 & 0.8992647177 \\
42 & -0.2514475296 \\
43 & 0.3794747138 \\
44 & -0.8612776970 \\
45 & 0.2031789568 \\
46 & -0.4292498504 \\
47 & 0.8992647177 \\
48 & -0.2514475296 \\
49 & 0.3794747138 \\
\end{array}
\]
Table 3: Chebyshev Coefficients - Continued (e)

\[ xe^{-x}Ei(x) = \sum_{k=0}^{47} 'A_kT_k(t), \quad t = (x - 22)/10, \quad (12 \leq x \leq 32) \]
<table>
<thead>
<tr>
<th>k</th>
<th>( A_k )</th>
<th>367</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2117028640 4369868632 9789991614E 01</td>
<td>23</td>
</tr>
<tr>
<td>1</td>
<td>-0.3204327273 7548579499 0618303177E-01</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>0.8891732077 3531683589 0184200335E-02</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>-0.2507952805 1892993708 8352422063E-02</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>0.7202789465 9598754887 576092487E-03</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>-0.2103490058 5011305342 353141256E-03</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>0.6205732318 2769321658 885773084E-04</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>-0.1826567649 8167026544 9155689733E-04</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>0.5270567515 2893637580 7788296811E-05</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>-0.1459665647 6199457532 3066719367E-05</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>0.8891732077 3531683589 0184200335E-02</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td>-0.2507952805 1892993708 8352422063E-02</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>0.7202789465 9598754887 576092487E-03</td>
<td>35</td>
</tr>
<tr>
<td>13</td>
<td>-0.2103490058 5011305342 353141256E-03</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>0.6205732318 2769321658 885773084E-04</td>
<td>37</td>
</tr>
<tr>
<td>15</td>
<td>-0.1826567649 8167026544 9155689733E-04</td>
<td>38</td>
</tr>
<tr>
<td>16</td>
<td>0.5270567515 2893637580 7788296811E-05</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>-0.1459665647 6199457532 3066719367E-05</td>
<td>40</td>
</tr>
<tr>
<td>18</td>
<td>0.8891732077 3531683589 0184200335E-02</td>
<td>41</td>
</tr>
<tr>
<td>19</td>
<td>-0.2507952805 1892993708 8352422063E-02</td>
<td>42</td>
</tr>
<tr>
<td>20</td>
<td>0.7202789465 9598754887 576092487E-03</td>
<td>43</td>
</tr>
<tr>
<td>21</td>
<td>-0.2103490058 5011305342 353141256E-03</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>0.6205732318 2769321658 885773084E-04</td>
<td>45</td>
</tr>
<tr>
<td>23</td>
<td>-0.1826567649 8167026544 9155689733E-04</td>
<td>46</td>
</tr>
<tr>
<td>24</td>
<td>0.5270567515 2893637580 7788296811E-05</td>
<td>47</td>
</tr>
</tbody>
</table>

---

```lisp
| package DFSFUN DoubleFloatSpecialFunctions |

```
Table 3: Chebyshev Coefficients - Continued (f)

\[ xe^{-x}Ei(x) = \sum_{k=0}^{46} A_k T_k(t), \quad t = (64/x) - 1, \quad (32 \leq x < \infty) \]
package DFSFUN DoubleFloatSpecialFunctions 

Ei6(y:OPR):OPR ==
  infinite? y => 1
  x:R:=retract(y)
  m:R:=64.0::R/x-1.0::R
  t:R:=acos(m::R)::R
  t01:= 0.20328439457961699087873844202E0::R*cos(0.0::R)::R/2.0::R
  t02:=t01+0.166992045203136285147618434339E-01::R*cos(t)
  t03:=t02+0.28452847243613468074248999985325E-03::R*cos(2.0::R*t)
  t04:=t03+0.75639443585162064894876693854E-05::R*cos(3.0::R*t)
  t05:=t04+0.279897128945085915750484318090E-06::R*cos(4.0::R*t)
  t06:=t05+0.135790182853453106952556392593E-07::R*cos(5.0::R*t)
  t07:=t06+0.834359620204046925585610289412E-09::R*cos(6.0::R*t)
  t08:=t07+0.637097172764024843827524337306E-10::R*cos(7.0::R*t)
  t09:=t08+0.600724760881186123576083084850E-11::R*cos(8.0::R*t)
  t10:=t09+0.702287614767735975092615688E-12::R*cos(9.0::R*t)
  t11:=t10+0.10183026737036876930667322152E-12::R*cos(10.0::R*t)
  t12:=t11+0.17618129034308800404066741554E-13::R*cos(11.0::R*t)
  t13:=t12+0.325082861423536069492070674E-14::R*cos(12.0::R*t)
  t14:=t13+0.507177002550581867881479300685E-15::R*cos(13.0::R*t)
t15 := t14 + 0.16651773870432942985203696E-16 :: R*cos(14.0 :: R*t)
t16 := t15 - 0.316675389079751440072289978777339E-18 :: R*cos(15.0 :: R*t)
t17 := t16 - 0.15884037636641415154823134074E-16 :: R*cos(16.0 :: R*t)
t18 := t17 - 0.41755132561380188896626550563E-17 :: R*cos(17.0 :: R*t)
t19 := t18 - 0.28923477497071418820286862358E-18 :: R*cos(18.0 :: R*t)
t20 := t19 - 0.2800625903396608072899787777339E-18 :: R*cos(19.0 :: R*t)
t21 := t20 + 0.132293863953270891405320036957E-16 :: R*cos(20.0 :: R*t)
t22 := t21 + 0.18044744417730199585334811191E-19 :: R*cos(21.0 :: R*t)
t23 := t22 - 0.79053840865226165662021080364E-20 :: R*cos(22.0 :: R*t)
t24 := t23 - 0.443571136636957344718167314045E-20 :: R*cos(23.0 :: R*t)
t25 := t24 - 0.426410399497810261760579779746E-21 :: R*cos(24.0 :: R*t)
t26 := t25 + 0.392010176693714390725625388636E-21 :: R*cos(25.0 :: R*t)
t27 := t26 + 0.152737805134396364447280486402E-21 :: R*cos(26.0 :: R*t)
t28 := t27 - 0.102484952704949060786953149788E-22 :: R*cos(27.0 :: R*t)
t29 := t28 - 0.2134907874771089379489042873231E-22 :: R*cos(28.0 :: R*t)
t30 := t29 - 0.323913947516023687614279789345E-23 :: R*cos(29.0 :: R*t)
t31 := t30 - 0.21421837622964597029624935934E-23 :: R*cos(30.0 :: R*t)
t32 := t31 - 0.82346094196189955316920738151E-24 :: R*cos(31.0 :: R*t)
t33 := t32 - 0.152465282962067210811495038147E-24 :: R*cos(32.0 :: R*t)
t34 := t33 + 0.137820828248824401290438126477E-24 :: R*cos(33.0 :: R*t)
t35 := t34 - 0.213131120142873706791513005998E-26 :: R*cos(34.0 :: R*t)
t36 := t35 + 0.20126496518713266859213006507E-25 :: R*cos(35.0 :: R*t)
t37 := t36 - 0.199553566205637402320607178286E-26 :: R*cos(36.0 :: R*t)
t38 := t37 - 0.279899581220179711426020884464E-26 :: R*cos(37.0 :: R*t)
t39 := t38 - 0.553451183050025949784942560E-27 :: R*cos(38.0 :: R*t)
t40 := t39 - 0.388499542268455253129749000696E-27 :: R*cos(39.0 :: R*t)
t41 := t40 - 0.112130440723307012540043264712E-27 :: R*cos(40.0 :: R*t)
t42 := t41 - 0.55665822661654988057823816866E-28 :: R*cos(41.0 :: R*t)
t43 := t42 - 0.20454262124651357628865878722E-28 :: R*cos(42.0 :: R*t)
t44 := t43 - 0.845381406484393809437361193598E-29 :: R*cos(43.0 :: R*t)
t45 := t44 - 0.3565755152101526590791715785E-29 :: R*cos(44.0 :: R*t)
t46 := t45 - 0.13365423477975180195777206E-29 :: R*cos(45.0 :: R*t)
t47 := t46 - 0.60621426532093450578786528630E-30 :: R*cos(46.0 :: R*t)

Table 4: Function Values of the Associated Functions

---
<table>
<thead>
<tr>
<th>$x$</th>
<th>$t = -(20/x) - 1$</th>
<th>$xe^{-x}Ei(x)$</th>
<th>$x = (x + 7)/3$</th>
<th>$xe^{-x}Ei(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>-1.000</td>
<td>0.1000000000 0000000000 0000000000 E 01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-160</td>
<td>-0.875</td>
<td>0.9938266956 7406127387 8797850088 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-80</td>
<td>-0.750</td>
<td>0.9878013330 9428877356 4522608410 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-53 1/3</td>
<td>-0.625</td>
<td>0.9819162901 4319443961 7735426105 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-40</td>
<td>-0.500</td>
<td>0.9761646031 8514305080 8000604060 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-32</td>
<td>-0.375</td>
<td>0.9705398840 7466392046 2584664361 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-26 2/3</td>
<td>-0.250</td>
<td>0.9650362511 2337703576 3536593528 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-22 6/7</td>
<td>-0.125</td>
<td>0.9564827110 7936727616 5478970820 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-20</td>
<td>-0.000</td>
<td>0.9543709099 1921683397 5195829433 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-17 7/9</td>
<td>0.125</td>
<td>0.9491994907 7974574460 6445346803 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>0.250</td>
<td>0.9441296577 3690297898 4149471583 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-14 6/11</td>
<td>0.375</td>
<td>0.9391573444 1928412346 0424240998 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-13 1/3</td>
<td>0.500</td>
<td>0.9342787466 5341016080 3097580165 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-12 4/13</td>
<td>0.625</td>
<td>0.9294902984 9721403772 5319679042 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-11 3/7</td>
<td>0.750</td>
<td>0.9247865111 2384169605 599385492 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10 2/3</td>
<td>0.875</td>
<td>0.9201706542 4944567620 2148012149 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>1.000</td>
<td>0.9156333393 9788081876 0698157666 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10.000</td>
<td>-1.000</td>
<td>0.9156333393 9788081876 0698157666 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-9.625</td>
<td>-0.875</td>
<td>0.9128444614 6799341885 6575662217 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-9.250</td>
<td>-0.750</td>
<td>0.9098627515 2542413937 895427597 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8.875</td>
<td>-0.625</td>
<td>0.9066672706 5475388033 4995756418 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8.500</td>
<td>-0.500</td>
<td>0.9032339019 7320784141 4682926135 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8.125</td>
<td>-0.375</td>
<td>0.8995347176 8847383630 1415777697 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.750</td>
<td>-0.250</td>
<td>0.8955371870 8753915717 9475513219 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.375</td>
<td>-0.125</td>
<td>0.8912031763 2125431626 708746258 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.000</td>
<td>-0.000</td>
<td>0.8864876725 3642932589 3993846569 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.625</td>
<td>0.125</td>
<td>0.8813371384 6821020039 4305706270 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.250</td>
<td>0.250</td>
<td>0.8756873647 8846593227 6462155532 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5.875</td>
<td>0.375</td>
<td>0.8694606294 5411341030 2047153364 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5.500</td>
<td>0.500</td>
<td>0.8625618846 9070142209 0918986586 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5.125</td>
<td>0.625</td>
<td>0.854875538 9019954239 2425567234 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.750</td>
<td>0.750</td>
<td>0.8462482991 0358736117 1665798810 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.375</td>
<td>0.875</td>
<td>0.8364987545 5629874174 2152267582 E 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.000</td>
<td>1.000</td>
<td>0.8253825996 042233240 8183035504 E 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[
\begin{array}{|c|c|c|}
\hline
x & t = x/4 & [\text{Ei}(x) - \log|x| - \gamma]/x \\
\hline
-4.0 & -1.000 & 0.4918223446 0781809647 9962798267 E 00 \\
-3.5 & -0.875 & 0.5248425066 4412835691 8258753311 E 00 \\
-3.0 & -0.750 & 0.5629587782 2127986318 8086024270 E 00 \\
-2.5 & -0.625 & 0.6073685258 5838306451 4266925640 E 00 \\
-2.0 & -0.500 & 0.6596316780 8476964479 5492023380 E 00 \\
-1.5 & -0.375 & 0.7218002369 4421992965 7623030310 E 00 \\
-1.0 & -0.250 & 0.7965995992 9705313428 3675865540 E 00 \\
-0.5 & -0.125 & 0.8876841582 3549675287 2151815870 E 00 \\
0.0 & 0.000 & 0.0000000000 0000000000 0000000000 E 01 \\
0.5 & 0.125 & 0.1140302841 0431720574 6248768807 E 01 \\
1.0 & 0.250 & 0.1317902151 4544038948 6008844242 E 01 \\
1.5 & 0.375 & 0.1545736450 7467337302 4859074039 E 01 \\
2.0 & 0.500 & 0.1841935755 2702059966 7788045934 E 01 \\
2.5 & 0.625 & 0.2232103799 1211651144 5340564235 E 01 \\
3.0 & 0.750 & 0.275268205 6852580020 0219289740 E 01 \\
3.5 & 0.875 & 0.345821531 9301241243 7300898811 E 01 \\
4.0 & 1.000 & 0.4416841111 0086991358 0118598686 E 01 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
x & t = (x - 8)/4 & xe^{-x}\text{Ei}(x) \\
\hline
4.0 & -1.000 & 0.1438208031 4544827847 0968670330 E 01 \\
4.5 & -0.875 & 0.1396419029 6297460710 0674523183 E 01 \\
5.0 & -0.750 & 0.1353831277 4552859779 0189174047 E 01 \\
5.5 & -0.625 & 0.131443565 7421192454 1219816991 E 01 \\
6.0 & -0.500 & 0.127883860 4895616189 234099578 E 01 \\
6.5 & -0.375 & 0.1248391155 0017014864 0741941387 E 01 \\
7.0 & -0.250 & 0.1222408052 3605310590 365846622 E 01 \\
7.5 & -0.125 & 0.1200421499 5996307864 3879158950 E 01 \\
8.0 & -0.000 & 0.118184798 9872079731 739362644 E 01 \\
8.5 & 0.125 & 0.1166126525 8117484943 9918142965 E 01 \\
9.0 & 0.250 & 0.1152759208 7089248132 2396814952 E 01 \\
9.5 & 0.375 & 0.114132475 9526242015 5338560641 E 01 \\
10.0 & 0.500 & 0.1131470204 7341077803 4051681355 E 01 \\
10.5 & 0.625 & 0.1122915570 0177606064 2888630755 E 01 \\
11.0 & 0.750 & 0.1115430938 9980384416 4779434229 E 01 \\
11.5 & 0.875 & 0.1108832926 3050773058 6855234934 E 01 \\
12.0 & 1.000 & 0.1102974544 9067590726 7241234953 E 01 \\
\hline
\end{array}
\]
<table>
<thead>
<tr>
<th>$x$</th>
<th>$t = (x - 22)/10$</th>
<th>$xe^{-x}Ei(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.00</td>
<td>-1.000</td>
<td>0.1102974544 9067590726 7241234952 E 01</td>
</tr>
<tr>
<td>13.25</td>
<td>-0.875</td>
<td>0.1090844898 2154756926 6468614954 E 01</td>
</tr>
<tr>
<td>14.50</td>
<td>-0.750</td>
<td>0.1081351395 7351912850 6346643795 E 01</td>
</tr>
<tr>
<td>15.75</td>
<td>-0.625</td>
<td>0.1073701384 1997572371 2157900374 E 01</td>
</tr>
<tr>
<td>17.00</td>
<td>-0.500</td>
<td>0.1067393691 9585378312 9572196197 E 01</td>
</tr>
<tr>
<td>18.25</td>
<td>-0.375</td>
<td>0.1062096608 6221502426 8372647556 E 01</td>
</tr>
<tr>
<td>19.50</td>
<td>-0.250</td>
<td>0.1057581342 1587250319 5393949410 E 01</td>
</tr>
<tr>
<td>20.75</td>
<td>-0.125</td>
<td>0.1053684451 2894094408 2102194964 E 01</td>
</tr>
<tr>
<td>22.00</td>
<td>-0.000</td>
<td>0.1050285719 6851897941 1780664532 E 01</td>
</tr>
<tr>
<td>23.25</td>
<td>0.125</td>
<td>0.1047294551 7053248581 4923655913 E 01</td>
</tr>
<tr>
<td>24.50</td>
<td>0.250</td>
<td>0.1044641267 9046436368 9761075289 E 01</td>
</tr>
<tr>
<td>25.75</td>
<td>0.375</td>
<td>0.1042271337 923202388 5710928048 E 01</td>
</tr>
<tr>
<td>27.00</td>
<td>0.500</td>
<td>0.1040141438 3230104381 3713899754 E 01</td>
</tr>
<tr>
<td>28.25</td>
<td>0.625</td>
<td>0.1038216700 3601458768 0056548394 E 01</td>
</tr>
<tr>
<td>29.50</td>
<td>0.750</td>
<td>0.1036468726 2924118457 5154685419 E 01</td>
</tr>
<tr>
<td>30.75</td>
<td>0.875</td>
<td>0.1034874149 8964796947 2990938590 E 01</td>
</tr>
<tr>
<td>32.00</td>
<td>1.000</td>
<td>0.1033413564 2162410494 3493552567 E 01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$t = (64/x) - 1$</th>
<th>$xe^{-x}Ei(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>-1.000</td>
<td>0.100000000 000000000 0000000001 E 01</td>
</tr>
<tr>
<td>512</td>
<td>-0.875</td>
<td>0.100196079 9450711925 3133746847 E 01</td>
</tr>
<tr>
<td>256</td>
<td>-0.750</td>
<td>0.100393713 0905698627 8800978297 E 01</td>
</tr>
<tr>
<td>170 2/3</td>
<td>-0.625</td>
<td>0.100592927 5692929112 9466030932 E 01</td>
</tr>
<tr>
<td>128</td>
<td>-0.500</td>
<td>0.100793752 408140182 8177682169 E 01</td>
</tr>
<tr>
<td>102 2/5</td>
<td>-0.375</td>
<td>0.100996217 7406449755 7436754557 E 01</td>
</tr>
<tr>
<td>85 1/3</td>
<td>-0.250</td>
<td>0.101200354 5332988482 0186466702 E 01</td>
</tr>
<tr>
<td>73 1/7</td>
<td>-0.125</td>
<td>0.101406194 9696971331 4594232935 E 01</td>
</tr>
<tr>
<td>64</td>
<td>-0.000</td>
<td>0.101613772 3494325321 7035710083 E 01</td>
</tr>
<tr>
<td>56 8/9</td>
<td>0.125</td>
<td>0.101823121 1884832696 8233701714 E 01</td>
</tr>
<tr>
<td>51 1/5</td>
<td>0.250</td>
<td>0.102034277 2930783774 8721782980 E 01</td>
</tr>
<tr>
<td>46 6/11</td>
<td>0.375</td>
<td>0.102247277 8405420595 9127536479 E 01</td>
</tr>
<tr>
<td>42 2/3</td>
<td>0.500</td>
<td>0.102462161 4681078391 0118780424 E 01</td>
</tr>
<tr>
<td>39 5/13</td>
<td>0.625</td>
<td>0.102679898 3709028524 5098451082 E 01</td>
</tr>
<tr>
<td>36 4/7</td>
<td>0.750</td>
<td>0.102897740 4105808008 6337843509 E 01</td>
</tr>
<tr>
<td>34 2/15</td>
<td>0.875</td>
<td>0.103118521 2364659263 5587584663 E 01</td>
</tr>
<tr>
<td>32</td>
<td>1.000</td>
<td>0.103341356 4216241049 4349355256 E 01</td>
</tr>
</tbody>
</table>
The Fresnel Integral[?, ?]

The Fresnel function is

\[ C(x) - iS(x) = \int_0^x \exp(-i\pi t^2/2) \, dt \]

We compare Axiom’s results to Pearcey’s tables which show the fresnel results to 6 decimal places. Computation of these values requires floats as the range quickly exceeds DoubleFloat. In each decade of the range we increase the number of terms by a factor of 10. So we compute with 10 terms in the range 0.0-10.0, 100 terms in 10.0-20.0, etc.

fresnelC

The fresnelC is the real portion of the Fresnel integral, \( C(u) \), is defined as:

\[ C(\sqrt{2x/\pi}) = \frac{1}{2} \int_0^x J_{-\frac{1}{2}}(t) \, dt = \frac{1}{\sqrt{(2\pi)}} \int_0^x \cos(t) \, dt \]

where \( J_{-\frac{1}{2}}(t) \) is the Bessel function of the first kind of order \( -\frac{1}{2} \).

This is related to the better known definition of \( C(u) \), namely:

\[ C(u) = \int_0^u \cos \frac{\pi t^2}{2} \, dt \]

where \( x = \pi u^2/2 \), or \( u = (2x/\pi)^{1/2} \)

fresnelC is an analytic function of \( z \) with \( z=0 \) as a two-sheeted branch point. Along the positive real axis the real definition gives:

\[ C(0) = 0 \]

\[ \lim_{x \to +\infty} C(x) = \frac{1}{2} \]

The asymptotic behavior of the function in the corner \( |\text{arc } z| \leq \pi - \epsilon \), (\( \epsilon > 0 \)), for \( |z| \gg 1 \) is given by

\[ C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}} \left( 1 - \frac{1 \cdot 3}{(2z)^2} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{(2z)^4} - \cdots \right) - \frac{\cos z}{\sqrt{2\pi z}} \left( \frac{1}{(2z)} - \frac{1 \cdot 3 \cdot 5}{(2z)^3} + \cdots \right) \]

(Note: Pearcey has a sign error for the second term ([?],p7)

The first approximation is

\[ C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}} \]

Axiom uses the power series at the zero point:

\[ C(z) = \sqrt{\frac{2z}{\pi}} \sum_{k=0}^{n} \frac{(-1)^k z^{2k}}{(4k+1)(2k)!} \]
fresnelC(z:F):F ==
z < 0 => error "fresnelC not defined for negative argument"
z = 0 => 0
n:PI:= 100
sqrt((2.0/pi()$F)*z)*_
reduce(_+,[(-1)**k*z**(2*k)/(factorial(2*k)*(4*k+1))_
for k in 0..n])$LF

fresnelS

The fresnelS is the complex portion of the Fresnel integral, S(u), is defined as:

\[
S(\sqrt{2x/\pi}) = \frac{1}{2} \int_0^x J_{\frac{1}{2}}(t) \, dt = \frac{1}{\sqrt{(2\pi)}} \int_0^x \frac{\sin(t)}{\sqrt{t}} \, dt
\]

where \(J_{\frac{1}{2}}(t)\) is the Bessel function of the first kind of order \(\frac{1}{2}\).

This is related to the better known definition of \(S(u)\), namely:

\[
S(u) = \int_0^u \sin \frac{\pi t^2}{2} \, dt
\]

where \(x = \pi u^2/2\), or \(u = (2x/\pi)^{1/2}\)

fresnelS is an analytic function of \(z\) with \(z=0\) as a two-sheeted branch point. Along the positive real axis the real definition gives:

\[
S(0) = 0
\]

\[
\lim_{x \to +\infty} S(x) = \frac{1}{2}
\]

The asymptotic behavior of the function in the corner \(|\text{arc } z| \leq \pi - \epsilon, (\epsilon > 0)\), for \(|z| \gg 1\) is given by

\[
S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}} \left( 1 - \frac{1 \cdot 3}{(2z)^2} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{(2z)^4} - \cdots \right) - \frac{\sin z}{\sqrt{2\pi z}} \left( \frac{1}{(2z)} - \frac{1 \cdot 3 \cdot 5}{(2z)^3} + \cdots \right)
\]

The first approximation is

\[
S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}}
\]
Axiom uses the power series at the zero point:

\[ S(z) = \sqrt{\frac{2z}{\pi}} \sum_{k=0}^{n} \frac{(-1)^k \pi^{2k+1}}{(4k+3)(2k+1)!} \]

--- package DFSFUN DoubleFloatSpecialFunctions ---

fresnelS(z:F):F ==
  z < 0 => error "fresnelS not defined for negative argument"
  z = 0 => 0
  n:PI:= 100
  sqrt((2.0/pi()$F)*z)*_ reduce(_+,
    [(-1)**k*(z**(2*k+1))/(factorial(2*k+1)*(4*k+3))
    for k in 0..n]$LF

--- package DFSFUN DoubleFloatSpecialFunctions ---

polygamma(k,z) == CPSI(k, z)$Lisp
polygamma(k,x) == RPSI(k, x)$Lisp

logGamma z == CLNGAMMA(z)$Lisp
logGamma x == RLNGAMMA(x)$Lisp

besselJ(v,z) == CBESSELJ(v,z)$Lisp
besselJ(n,x) == RBESSELJ(n,x)$Lisp

besselI(v,z) == CBESSELI(v,z)$Lisp
besselI(n,x) == RBESSELI(n,x)$Lisp

hypergeometric0F1(a,z) == CHYPER0F1(a, z)$Lisp
hypergeometric0F1(n,x) == retract hypergeometric0F1(n::C, x::C)

-- All others are defined in terms of these.
digamma x == polygamma(0, x)
digamma z == polygamma(0, z)

Beta(x,y) == Gamma(x)*Gamma(y)/Gamma(x+y)
Beta(w,z) == Gamma(w)*Gamma(z)/Gamma(w+z)

fuzz := (10::R)**(-7)

import IntegerRetractions(R)
import IntegerRetractions(C)
besselY(n,x) ==
    if integer? n then n := n + fuzz
    vp := n * pi()$R
    (cos(vp) * besselJ(n,x) - besselJ(-n,x) )/sin(vp)

besselY(v,z) ==
    if integer? v then v := v + fuzz::C
    vp := v * pi()$C
    (cos(vp) * besselJ(v,z) - besselJ(-v,z) )/sin(vp)

besselK(n,x) ==
    if integer? n then n := n + fuzz
    p := pi()$R
    vp := n*p
    ahalf:= 1/(2::R)
    p * ahalf * ( besselI(-n,x) - besselI(n,x) )/sin(vp)

besselK(v,z) ==
    if integer? v then v := v + fuzz::C
    p := pi()$C
    vp := v*p
    ahalf:= 1/(2::C)
    p * ahalf * ( besselI(-v,z) - besselI(v,z) )/sin(vp)

airyAi x ==
    ahalf := recip(2::R)::R
    athird := recip(3::R)::R
    eta := 2 * athird * (-x) ** (3*ahalf)
    (-x)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))

airyAi z ==
    ahalf := recip(2::C)::C
    athird := recip(3::C)::C
    eta := 2 * athird * (-z) ** (3*ahalf)
    (-z)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))

airyBi x ==
    ahalf := recip(2::R)::R
    athird := recip(3::R)::R
    eta := 2 * athird * (-x) ** (3*ahalf)
    (-x*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )

airyBi z ==
    ahalf := recip(2::C)::C
    athird := recip(3::C)::C
    eta := 2 * athird * (-z) ** (3*ahalf)
    (-z*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )
package DBLRESP DoubleResultantPackage

-- DoubleResultantPackage.input --

)set break resume
)sys rm -f DoubleResultantPackage.output
)spool DoubleResultantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DoubleResultantPackage
--E 1

)spool
)lisp (bye)

---

-- DoubleResultantPackage.help --

====================================================================
DoubleResultantPackage examples
====================================================================

This package provides functions for computing the residues of a function on an algebraic curve.

See Also:
o )show DoubleResultantPackage

---
DoubleResultantPackage (DBLRESP)

Exports:
doubleResultant

— package DBLRESP DoubleResultantPackage —

)abbrev package DBLRESP DoubleResultantPackage
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 July 1990
++ Description:
++ This package provides functions for computing the residues
++ of a function on an algebraic curve.

DoubleResultantPackage(F, UP, UPUP, R): Exports == Implementation where
F : Field
UP : UnivariatePolynomialCategory F
UPUP: UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F, UP, UPUP)

RF ==> Fraction UP
UP2 ==> SparseUnivariatePolynomial UP
UP3 ==> SparseUnivariatePolynomial UP2

Exports ==> with
doubleResultant: (R, UP -> UP) -> UP
   ++ doubleResultant(f, ') returns p(x) whose roots are
   ++ rational multiples of the residues of f at all its
   ++ finite poles. Argument ' is the derivation to use.

Implementation ==> add
import CommuteUnivariatePolynomialCategory(F, UP, UP2)
import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

UP22 : UP  -> UP
UP23 : UPUP -> UP3
\begin{verbatim}
remove0: UP \rightarrow UP -- removes the power of x dividing p

remove0 p ==
  primitivePart((p exquo monomial(1, minimumDegree p))::UP)

UP22 p ==
  map(x\rightarrow x::UP, p)$UnivariatePolynomialCategoryFunctions2(F,UP,UP,UP2)

UP23 p ==
  map(x\rightarrow UP22(retract(x)@UP), p)
    $UnivariatePolynomialCategoryFunctions2(RF, UPUP, UP2, UP3)

doubleResultant(h, derivation) ==
  cd := splitDenominator lift h
  d := (cd.den exquo (g := gcd(cd.den, derivation(cd.den))))::UP
  r := swap primitivePart swap resultant(UP23(cd.num)
    - ((monomial(1, 1)$UP :: UP2) * UP22(g * derivation d))::UP3,
      UP23 definingPolynomial())
  remove0 resultant(r, UP22 d)

package DRAWCX DrawComplex

\end{verbatim}
DrawComplex (DRAWCX)

Exports:
setImagSteps setRealSteps drawComplex drawComplexVectorField setClipValue

— package DRAWCX DrawComplex —

)abbrev package DRAWCX DrawComplex
++ Description:
++ \axiomType{DrawComplex} provides some facilities
++ for drawing complex functions.

DrawComplex(): Exports == Implementation where
C ==> Complex DoubleFloat
S ==> Segment DoubleFloat
PC ==> Record(rr:SF, th:SF)
INT ==> Integer
SF ==> DoubleFloat
NNI ==> NonNegativeInteger
VIEW3D ==> ThreeDimensionalViewport
ARRAY2 ==> TwoDimensionalArray
Exports == with
drawComplex: (C -> C,S,S,Boolean) -> VIEW3D
to drawComplex(f,rRange,iRange,arrows?)
++ draws a complex function as a height field.
++ It uses the complex norm as the height and the complex
++ argument as the color.
++ It will optionally draw arrows on the surface indicating the direction
++ of the value.
++ Sample call:
++ \spad{f z == exp(1/z)}
++ \spad{drawComplex(f, 0.3..3, 0..2*%pi, false)}
++ Parameter descriptions:
++ f: the function to draw
++ rRange : the range of the real values
++ iRange : the range of imaginary values
++ arrows? : a flag indicating whether to draw the phase arrows for f
++ Call the functions \axiomFunFrom{setRealSteps}{DrawComplex} and
++ \axiomFunFrom{setImagSteps}{DrawComplex} to change the
++ number of steps used in each direction.
drawComplexVectorField: (C -> C,S,S) -> VIEW3D
++ drawComplexVectorField(f,rRange,iRange)
++ draws a complex vector field using arrows on the \spad{x--y} plane.
++ These vector fields should be viewed from the top by pressing the
++ "XY" translate button on the 3-d viewport control panel.
++ Sample call:
++ \spad{f z == sin z}
++ \spad{drawComplexVectorField(f, -2..2, -2..2)}
++ Parameter descriptions:
++ f: the function to draw
++ rRange : the range of the real values
++ iRange : the range of imaginary values
++ Call the functions \axiomFunFrom{setRealSteps}{DrawComplex} and
++ \axiomFunFrom{setImagSteps}{DrawComplex} to change the
++ number of steps used in each direction.
setRealSteps: INT -> INT
++ setRealSteps(i)
++ sets to i the number of steps to use in the real direction
++ when drawing complex functions. Returns i.
setImagSteps: INT -> INT
++ setImagSteps(i)
++ sets to i the number of steps to use in the imaginary direction
++ when drawing complex functions. Returns i.
setClipValue: SF -> SF
   ++ setClipValue(x)
   ++ sets to x the maximum value to plot when drawing complex functions. Returns x.
Implementation == add

-- relative size of the arrow head compared to the length of the arrow
arrowScale : SF := (0.125)::SF
arrowAngle: SF := pi()-pi()/(20::SF) -- angle of the arrow head
realSteps: INT := 11 -- the number of steps in the real direction
imagSteps: INT := 11 -- the number of steps in the imaginary direction
clipValue: SF := 10::SF -- the maximum length of a vector to draw

-- Add an arrow head to a line segment, which starts at 'p1', ends at 'p2',
-- has length 'len', and and angle 'arg'. We pass 'len' and 'arg' as
-- arguments since they were already computed by the calling program
makeArrow(p1:Point SF, p2:Point SF, len:SF, arg:SF):List List Point SF ==
c1 := cos(arg + arrowAngle)
s1 := sin(arg + arrowAngle)
c2 := cos(arg - arrowAngle)
s2 := sin(arg - arrowAngle)
p3 := point [p2.1 + c1*arrowScale*len, p2.2 + s1*arrowScale*len,
   p2.3, p2.4]
p4 := point [p2.1 + c2*arrowScale*len, p2.2 + s2*arrowScale*len,
   p2.3, p2.4]
[[p1, p2, p3], [p2, p4]]

-- clip a value in the interval (-clip...clip)
clipFun(x:SF):SF ==
   min(max(x, -clipValue), clipValue)

drawComplex(f, realRange, imagRange, arrows?) ==
delReal := (hi(realRange) - lo(realRange))/realSteps::SF
delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
funTable: ARRAY2(PC) :=
   new((realSteps::NNI)+1, (imagSteps::NNI)+1, [0,0]$PC)
real := lo(realRange)
for i in 1..realSteps+1 repeat
   imag := lo(imagRange)
   for j in 1..imagSteps+1 repeat
      z := f complex(real, imag)
      funTable(i,j) := [clipFun(sqrt norm z), argument(z)]$PC
      imag := imag + delImag
   end for
   real := real + delReal
end for
lp := empty()$(List List Point SF)
real := lo(realRange)
for i in 1..realSteps+1 repeat
   imag := lo(imagRange)
   lp := empty()$(List Point SF)
   for j in 1..imagSteps+1 repeat
      p := point [real, imag, funTable(i,j).rr, funTable(i,j).th]
lp := cons(p, lp)
imag := imag + delImag
real := real + delReal
llp := cons(lp, llp)

space := mesh(llp)$ThreeSpace
if arrows? then
    real := lo(realRange)
    for i in 1..realSteps+1 repeat
        imag := lo(imagRange)
        for j in 1..imagSteps+1 repeat
            arg := funTable(i,j).th
            p1 := point [real, imag, funTable(i,j).rr, arg]
            len := delReal*2.0::SF
            p2 := point [p1.1 + len*cos(arg), p1.2 + len*sin(arg),
                         p1.3, p1.4]
            arrow := makeArrow(p1, p2, len, arg)
            for a in arrow repeat
                curve(space, a)$ThreeSpace
        imag := imag + delImag
        real := real + delReal

makeViewport3D(space, "Complex Function")$VIEW3D

drawComplexVectorField(f, realRange, imagRange): VIEW3D ==
    -- compute the steps size of the grid
    delReal := (hi(realRange) - lo(realRange))/realSteps::SF
    delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
    -- create the space to hold the arrows
    space := create3Space()$ThreeSpace
    real := lo(realRange)
    for i in 1..realSteps+1 repeat
        imag := lo(imagRange)
        for j in 1..imagSteps+1 repeat
            z := f complex(real, imag)
            arg := argument z
            len := clipFun(sqrt norm z)
            p1 := point [real, imag, 0::SF, arg]
            scaleLen := delReal * len
            p2 := point [p1.1 + scaleLen*cos(arg), p1.2 + scaleLen*sin(arg),
                         0::SF, arg]
            arrow := makeArrow(p1, p2, scaleLen, arg)
            for a in arrow repeat
                curve(space, a)$ThreeSpace
        imag := imag + delImag
        real := real + delReal
-- draw the vector field
makeViewport3D(space, "Complex Vector Field")$VIEW3D

-- set the number of steps to use in the real direction
setRealSteps(n) ==
realSteps := n

-- set the number of steps to use in the imaginary direction
setImagSteps(n) ==
imagSteps := n

-- set the maximum value to plot
setClipValue clip ==
clipValue := clip

package DRAWHACK DrawNumericHack

— DrawNumericHack.input —

)set break resume
)sys rm -f DrawNumericHack.output
)spool DrawNumericHack.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawNumericHack
--E 1

)spool
)lisp (bye)
DrawNumericHack helps with drawing interfaces. DrawNumericHack provides a "coercion" from something of the form \( x = a..b \) where \( a \) and \( b \) are formal expressions to a binding of the form \( x = c..d \) where \( c \) and \( d \) are the numerical values of \( a \) and \( b \). This "coercion" fails if \( a \) and \( b \) contain symbolic variables, but is meant for expressions involving \( \%pi \).

Note that this package is meant for internal use only.

See Also:
- \( \text{show DrawNumericHack} \)
++ Hack for the draw interface. DrawNumericHack provides
++ a "coercion" from something of the form $x = a..b$ where $a$
++ and $b$ are
++ formal expressions to a binding of the form $x = c..d$ where $c$ and $d$
++ are the numerical values of $a$ and $b$. This "coercion" fails if
++ $a$ and $b$ contains symbolic variables, but is meant for expressions
++ involving $\pi$.
++ Note that this package is meant for internal use only.

DrawNumericHack(R:Join(OrderedSet,IntegralDomain,ConvertibleTo Float)):
  with coerce: SegmentBinding Expression R -> SegmentBinding Float
  ++ coerce(x = a..b) returns $x = c..d$ where $c$ and $d$ are the
  ++ numerical values of $a$ and $b$.
  == add
  coerce s ==
  map(numeric$Numeric(R),s)$SegmentBindingFunctions2(Expression R, Float)

— DRAWHACK.dotabb —

"DRAWHACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWHACK"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"DRAWHACK" -> "ALGEBRA"

———

package DROPT0 DrawOptionFunctions0

—— DrawOptionFunctions0.input ——

)set break resume
)sys rm -f DrawOptionFunctions0.output
)spool DrawOptionFunctions0.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawOptionFunctions0
--E 1

)spool
)lisp (bye)
This package has no description

See Also:

o )show DrawOptionFunctions0

Exports:

- adaptive
- clipBoolean
- coord
- curveColorPalette
- pointColorPalette
- ranges
- style
- title
- toScale
- tubePoints
- tubeRadius
- units
- var1Steps
- var2Steps
- viewpoint

)abbrev package DROPT0 DrawOptionFunctions0
++ Description:
++ This package has no description
-- The functions here are not in DrawOptions since they are not
-- visible to the interpreter.

DrawOptionFunctions0(): Exports == Implementation where
RANGE ==> List Segment Float
UNIT ==> List Float
PAL ==> Palette
POINT ==> Point(DoubleFloat)
SEG ==> Segment Float
SF ==> DoubleFloat
SPACE3 ==> ThreeSpace(DoubleFloat)
VIEWPT ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, deltaY:SF )
Exports ==> with
  adaptive: (List DrawOption, Boolean) -> Boolean
    ++ adaptive(l,b) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{adaptive}.
    ++ If the option does not exist the value, b is returned.
clipBoolean: (List DrawOption, Boolean) -> Boolean
  ++ clipBoolean(l,b) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{clipBoolean}.
  ++ If the option does not exist the value, b is returned.
viewpoint: (List DrawOption, VIEWPT) -> VIEWPT
  ++ viewpoint(l,ls) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{viewpoint}.
  ++ IF the option does not exist, the value ls is returned.
title: (List DrawOption, String) -> String
  ++ title(l,s) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{title}.
  ++ If the option does not exist the value, s is returned.
style: (List DrawOption, String) -> String
  ++ style(l,s) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{style}.
  ++ If the option does not exist the value, s is returned.
toScale: (List DrawOption, Boolean) -> Boolean
  ++ toScale(l,b) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{toScale}.
  ++ If the option does not exist the value, b is returned.
pointColorPalette: (List DrawOption,PAL) -> PAL
  ++ pointColorPalette(l,p) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{pointColorPalette}.
  ++ If the option does not exist the value, p is returned.
curveColorPalette: (List DrawOption,PAL) -> PAL
  ++ curveColorPalette(l,p) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{curveColorPalette}.
  ++ If the option does not exist the value, p is returned.
ranges: (List DrawOption, RANGE) -> RANGE
  ++ ranges(l,r) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{ranges}.
  ++ If the option does not exist the value, r is returned.
var1Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
  ++ var1Steps(l,n) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{var1Steps}.
++ If the option does not exist the value, n is returned.

var2Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
  ++ var2Steps(l,n) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{var2Steps}.
  ++ If the option does not exist the value, n is returned.

space: (List DrawOption) -> SPACE3
  ++ space(l) takes a list of draw options, l, and checks to see
  ++ if it contains the option \spad{space}. If the the option
  ++ doesn’t exist, then an empty space is returned.

tubePoints : (List DrawOption, PositiveInteger) -> PositiveInteger
  ++ tubePoints(l,n) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{tubePoints}.
  ++ If the option does not exist the value, n is returned.

tubeRadius : (List DrawOption, Float) -> Float
  ++ tubeRadius(l,n) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{tubeRadius}.
  ++ If the option does not exist the value, n is returned.

coord: (List DrawOption, (POINT->POINT)) -> (POINT->POINT)
  ++ coord(l,p) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{coord}.
  ++ If the option does not exist the value, p is returned.

units: (List DrawOption, UNIT) -> UNIT
  ++ units(l,u) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{units}.
  ++ If the option does not exist the value, u is returned.

Implementation ==> add

adaptive(l,s) ==
  (u := option(l, "adaptive"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

clipBoolean(l,s) ==
  (u := option(l, "clipBoolean"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

title(l, s) ==
  (u := option(l, "title"::Symbol)$DrawOptionFunctions1(String))
  case "failed" => s
  u::String

viewpoint(l, vp) ==
  (u := option(l, "viewpoint"::Symbol)$DrawOptionFunctions1(VIEWPT))
  case "failed" => vp
  u::VIEWPT

style(l, s) ==
  (u := option(l, "style"::Symbol)$DrawOptionFunctions1(String))
  case "failed" => s
u::String

toScale(l,s) ==
  (u := option(l, "toScale"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

pointColorPalette(l,s) ==
  (u := option(l, "pointColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
  case "failed" => s
  u::PAL

curveColorPalette(l,s) ==
  (u := option(l, "curveColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
  case "failed" => s
  u::PAL

ranges(l, s) ==
  (u := option(l, "ranges"::Symbol)$DrawOptionFunctions1(RANGE))
  case "failed" => s
  u::RANGE

space(l) ==
  (u := option(l, "space"::Symbol)$DrawOptionFunctions1(SPACE3))
  case "failed" => create3Space()$SPACE3
  u::SPACE3

var1Steps(l,s) ==
  (u := option(l, "var1Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger

var2Steps(l,s) ==
  (u := option(l, "var2Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger

tubePoints(l,s) ==
  (u := option(l, "tubePoints"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger

tubeRadius(l,s) ==
  (u := option(l, "tubeRadius"::Symbol)$DrawOptionFunctions1(Float))
  case "failed" => s
  u::Float

coord(l,s) ==
(u := option(l, "coord"::Symbol)$DrawOptionFunctions1(POINT->POINT))
case "failed" => s
u::(POINT->POINT)

units(l,s) ==
(u := option(l, "unit"::Symbol)$DrawOptionFunctions1(UNIT))
case "failed" => s
u::UNIT

---

DROPT0.dotabb

"DROPT0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT0"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DROPT0" -> "ALIST"

---

package DROPT1 DrawOptionFunctions1

--- DrawOptionFunctions1.input ---

)set break resume
)sys rm -f DrawOptionFunctions1.output
)spool DrawOptionFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawOptionFunctions1
--E 1

)spool
)lisp (bye)

---

--- DrawOptionFunctions1.help ---

================================================================================
DrawOptionFunctions1 examples
This package has no description

See Also:
  o )show DrawOptionFunctions1
++ is contained in the list of drawing options, \( l \), which is defined
++ by the draw command.
Implementation \( \implies \) add
\[
\text{option}(l, s) \Rightarrow \begin{cases} 
(u := \text{option}(l, s) \cup \text{Union}(\text{Any, "failed"})) \text{ case "failed"} \Rightarrow "\text{failed}"
\end{cases}
\text{retract}(u::\text{Any})$\text{AnyFunctions1}(S)
\]

---

— DROPT1.dotabb —

"DROPT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"DROPT1" -> "TYPE"

---

package D01AGNT d01AgentsPackage

— d01AgentsPackage.input —

)set break resume
)sys rm -f d01AgentsPackage.output
)spool d01AgentsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01AgentsPackage
--E 1

)spool
)lisp (bye)

---

— d01AgentsPackage.help —

d01AgentsPackage examples

d01AgentsPackage is a package of numerical agents to be used to investigate attributes of an input function so as to decide the measure of an appropriate numerical integration routine.

It contains functions `rangeIsFinite` to test the input range and `functionIsContinuousAtEndPoints` to check for continuity at the end points of the range.

See Also:
- `)show d01AgentsPackage`

---

**d01AgentsPackage (D01AGNT)**

```
D01AGNT
  FS

Exports:
  changeName  commaSeparate  df2st  functionIsContinuousAtEndPoints  functionIsOscillatory
  gethi       getlo       ldf2lst  problemPoints                   rangeIsFinite
  sdf2lst     singularitiesOf

--- package D01AGNT d01AgentsPackage ---
```

`)abbrev package D01AGNT d01AgentsPackage
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \texttt{\textbackslash axiomType\{d01AgentsPackage\}} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \texttt{\textbackslash axiomFun\{measure\}} of an appropriate numerical integration routine.
++ It contains functions \texttt{\textbackslash axiomFun\{rangeIsFinite\}} to test the input range and
++ \texttt{\textbackslash axiomFun\{functionIsContinuousAtEndPoints\}} to check for continuity at
++ the end points of the range.`
d01AgentsPackage(): E == I where

EF2 ==> ExpressionFunctions2
EFI ==> Expression Fraction Integer
FI ==> Fraction Integer
LEDF ==> List Expression DoubleFloat
KEDF ==> Kernel Expression DoubleFloat
EEDF ==> Equation Expression DoubleFloat
EDF ==> Expression DoubleFloat
PDF ==> Polynomial DoubleFloat
LDF ==> List DoubleFloat
SDF ==> Stream DoubleFloat
DF ==> DoubleFloat
F ==> Float
ST ==> String
LST ==> List String
SI ==> SingleInteger
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
OCEDF ==> OrderedCompletion Expression DoubleFloat
ECEFI ==> Equation OrderedCompletion Expression Fraction Integer
OCEFI ==> OrderedCompletion Expression Fraction Integer
OCFI ==> OrderedCompletion Fraction Integer
NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
INT ==> Integer
CTYPE ==> Union(continuous: "Continuous at the end points",
           lowerSingular: "There is a singularity at the lower end point",
           upperSingular: "There is a singularity at the upper end point",
           bothSingular: "There are singularities at both end points",
           notEvaluated: "End point continuity not yet evaluated")
RTYPE ==> Union(finite: "The range is finite",
           lowerInfinite: "The bottom of range is infinite",
           upperInfinite: "The top of range is infinite",
           bothInfinite: "Both top and bottom points are infinite",
           notEvaluated: "Range not yet evaluated")
STYPE ==> Union(str:SDF,
           notEvaluated:"Internal singularities not yet evaluated")
ATT ==> Record(endPointContinuity:CTYPE,
              singularitiesStream:STYPE,range:RTYPE)
ROA ==> Record(key:NIA,entry:ATT)

E ==> with

rangeIsFinite : NIA -> RTYPE
++ rangeIsFinite(args) tests the endpoints of \spad{args.range} for
++ infinite end points.
functionIsContinuousAtEndPoints: NIA -> CTYPE
++ functionIsContinuousAtEndPoints(args) uses power series limits
++ to check for problems at the end points of the range of \spad{args}.
getlo : SOCDF -> DF
++ getlo(x) gets the \axiomType{DoubleFloat} equivalent of
++ the first endpoint of the range \axiom{x}
gethi : SOCDF -> DF
++ gethi(x) gets the \axiomType{DoubleFloat} equivalent of
++ the second endpoint of the range \axiom{x}
functionIsOscillatory:NIA -> F
++ functionIsOscillatory(a) tests whether the function \spad{a.fn}
++ has many zeros of its derivative.
problemPoints: (EDF, Symbol, SOCDF) -> List DF
++ problemPoints(f,var,range) returns a list of possible problem points
++ by looking at the zeros of the denominator of the function if it
++ can be retracted to \axiomType{Polynomial DoubleFloat}.
singularitiesOf:NIA -> SDF
++ singularitiesOf(args) returns a list of potential
++ singularities of the function within the given range
df2st:DF -> String
++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
ldf2lst:LDF -> LST
++ ldf2lst(ln) coerces a List of \axiomType{DoubleFloat} to
++ \axiomType{List String}
sdf2lst:SDF -> LST
++ sdf2lst(ln) coerces a Stream of \axiomType{DoubleFloat} to
++ \axiomType{List String}
commaSeparate:LST -> ST
++ commaSeparate(l) produces a comma separated string from a
++ list of strings.
changeName:(Symbol,Symbol,Result) -> Result
++ changeName(s,t,r) changes the name of item \axiom{s} in \axiom{r}
++ to \axiom{t}.

I ==> ExpertSystemContinuityPackage add

import ExpertSystemToolsPackage
import ExpertSystemContinuityPackage

-- local functions
ocdf2ocefi : OCDF -> OCEFI
rangeOfArgument : (KEDF, NIA) -> DF
continuousAtPoint? : (EFI,EOCEFI) -> Boolean
rand:(SOCDF,INT) -> LDF
eval:(EDF,Symbol,LDF) -> LDF
numberOfSignChanges:LDF -> INT
rangeIsFiniteFunction:NIA -> RTYPE
functionIsContinuousAtEndPointsFunction:NIA -> CTYPE

changeName(s:Symbol,t:Symbol,r:Result):Result ==
a := remove!(s,r)$Result
a case Any =>
  insert!([t,a],r)$Result
r
commaSeparate(l:LST):ST ==
   empty?(l$LST) => ""
   -- one?(#(l)) => concat(l)$ST
   (#(l) = 1) => concat(l)$ST
   f := first(l)$LST
   t := [concat([", ",l.i]$ST for i in 2..#(l)]
   concat(f,concat(t)$ST)$ST

rand(seg:SOCDF,n:INT):LDF ==
   -- produced a sorted list of random numbers in the given range
   l:DF := getlo seg
   s:DF := (gethi seg) - 1
   seed:INT := random()$INT
   dseed:DF := seed :: DF
   r:LDF := [(((random(seed)$INT) :: DF)*s/dseed + l) for i in 1..n]
   sort(r)$LDF

eval(f:EDF,var:Symbol,l:LDF):LDF ==
   empty?(l$LDF) => [0$DF]
   ve := var::EDF
   retract(eval(f,equation(ve,u::EDF)$EEDF)$EDF)@DF for u in l

numberOfSignChanges(l:LDF):INT ==
   -- calculates the number of sign changes in a list
   a := 0$INT
   empty?(l$LDF) => 0
   for i in 2..# l repeat
      if negative?(l.i*l.(i-1)) then
         a := a + 1
   a

rangeOfArgument(k: KEDF, args:NIA): DF ==
   Args := copy args
   Args.fn := arg := first(argument(k)$KEDF)$LEDF
   functionIsContinuousAtEndPoints(Args) case continuous =>
      r:SOCDF := args.range
      low:EDF := (getlo r) :: EDF
      high:EDF := (gethi r) :: EDF
      eql := equation(a := args.var :: EDF, low)$EEDF
      eqh := equation(a, high)$EEDF
      e1 := (numeric(eval(arg,eql)$EDF)$Numeric(DF)) :: DF
      e2 := (numeric(eval(arg,eqh)$EDF)$Numeric(DF)) :: DF
      e2-e1
   0$DF

ocdf2ocefi(r:OCDF):OCEFI ==
   finite?(r)$OCDF => (edf2efi(((retract(r)@DF)$OCDF)::EDF))::OCEFI
   r pretend OCEFI
CHAPTER 5. CHAPTER D

continuousAtPoint?(f:EFI,e:EOCEFI):Boolean ==
  (l := limit(f,e)$PowerSeriesLimitPackage(FI,EFI)) case OCEFI =>
    finite?(l :: OCEFI)
  -- if the left hand limit equals the right hand limit, or if neither
  -- side has a limit at this point, the return type of limit() is
  -- Union(Ordered Completion Expression Fraction Integer,"failed")
  false

-- exported functions

rangeIsFiniteFunction(args:NIA): RTYPE ==
  rangeIsFinite(x) tests the endpoints of x.range for infinite
  end points.
  --
  -- [-inf, inf] => 4
  -- [ x , inf] => 3
  -- [-inf, x ] => 1
  -- [ x , y ] => 0
  fr:SI := (3::SI * whatInfinity(hi(args.range))$OCDF
            - whatInfinity(lo(args.range))$OCDF)
  fr = 0 => "The range is finite"
  fr = 1 => "The bottom of range is infinite"
  fr = 3 => "The top of range is infinite"
  fr = 4 => "Both top and bottom points are infinite"
  error("rangeIsFinite","this is not a valid range")$ErrorFunctions

rangeIsFinite(args:NIA): RTYPE ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s := coerce(t)@ATT
    s.range case notEvaluated =>
      s.range := rangeIsFiniteFunction(nia)
      r:ROA := [nia,s]
      insert!(r)$IntegrationFunctionsTable
      s.range
    s.range
    a:ATT := ["End point continuity not yet evaluated"],
    ["Internal singularities not yet evaluated"],
    e:=rangeIsFiniteFunction(nia]
    r:ROA := [nia,a]
    insert!(r)$IntegrationFunctionsTable
    e

functionIsContinuousAtEndPointsFunction(args:NIA):CTYPE ==
  v := args.var :: EFI :: OCEFI
  high:OCEFI := ocdf2ocdf(e(hi(args.range)))
  low:OCEFI := ocdf2ocdf(e(lo(args.range)))
  f := edf2efi(args.fn)
  l:Boolean := continuousAtPoint?(f,equation(v,low)$EOCEFI)
  h:Boolean := continuousAtPoint?(f,equation(v,high)$EOCEFI)
l and h => ["Continuous at the end points"]
l => ["There is a singularity at the upper end point"]
h => ["There is a singularity at the lower end point"]
["There are singularities at both end points"]

functionIsContinuousAtEndPoints(args:NIA): CTYPE ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s := coerce(t)@ATT
    s.endPointContinuity case notEvaluated =>
      s.endPointContinuity := functionIsContinuousAtEndPointsFunction(nia)
      r:ROA := [nia, s]
      insert!(r)$IntegrationFunctionsTable
    s.endPointContinuity
  a: ATT := [e := functionIsContinuousAtEndPointsFunction(nia),
              ["Internal singularities not yet evaluated"],
              ["Range not yet evaluated"]]
  r:ROA := [nia, a]
  insert!(r)$IntegrationFunctionsTable
e

functionIsOscillatory(a:NIA): F ==
  args := copy a
  k := tower(numerator args.fn)$EDF
  p:F := pi()$F
  for i in 1..# k repeat
    is?(ker := k.i, sin :: Symbol) =>
      ra := convert(rangeOfArgument(ker, args))@F
      ra > 2*p => return (ra/p)
    is?(ker, cos :: Symbol) =>
      ra := convert(rangeOfArgument(ker, args))@F
      ra > 2*p => return (ra/p)
  l:LDF := rand(args.range, 30)
  l := eval(args.fn, args.var, l)
  numberOfSignChanges(l) :: F

singularitiesOf(args:NIA): SDF ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s: ATT := coerce(t)@ATT
    p: STYPE := s.singularitiesStream
    p case str => p.str
    e:SDF := singularitiesOf(nia.fn, [nia.var], nia.range)
    if not empty?(e) then
      if less?(e, 10)$SDF then extend(e, 10)$SDF
      s.singularitiesStream := [e]
    r:ROA := [nia, s]
    insert!(r)$IntegrationFunctionsTable
e := singularitiesOf(nia.fn,[nia.var],nia.range)
if not empty?(e) then
  if less?(e,10)$SDF then extend(e,10)$SDF
a:ATT := [["End point continuity not yet evaluated"],[e],
  ["Range not yet evaluated"]]
r:ROA := [nia,a]
insert!(r)$IntegrationFunctionsTable

— D01AGNT.dotabb —

"D01AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D01AGNT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D01AGNT" -> "FS"

package D01WGTS d01WeightsPackage

— d01WeightsPackage.input —

)set break resume
)sys rm -f d01WeightsPackage.output
)spool d01WeightsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01WeightsPackage
--E 1

)spool
)lisp (bye)

— d01WeightsPackage.help —

====================================================================
d01WeightsPackage is a package for functions used to investigate whether a function can be divided into a simpler function and a weight function. The types of weights investigated are those giving rise to end-point singularities of the algebraico-logarithmic type, and trigonometric weights.

See Also:
o )show d01WeightsPackage

---

Exports:
expressHasAlgebraicWeight expressHasLogarithmicWeights expressHasWeightCosWXorSinWX

--- package D01WGTS d01WeightsPackage ---

)abbrev package D01WGTS d01WeightsPackage
++ Author: Brian Dupee
++ Date Created: July 1994
++ Date Last Updated: January 1998
++ Description:
++ \axiom{d01WeightsPackage} is a package for functions used to investigate whether a function can be divided into a simpler function and a weight function. The types of weights investigated are those giving rise to end-point singularities of the algebraico-logarithmic type, and trigonometric weights.
++
++ d01WeightsPackage(): E == I where
LEDF ==> List Expression DoubleFloat
KEDF ==> Kernel Expression DoubleFloat
LKEDF ==> List Kernel Expression DoubleFloat
EDF ==> Expression DoubleFloat
PDF ==> Polynomial DoubleFloat
FI ==> Fraction Integer
LDF ==> List DoubleFloat
DF ==> DoubleFloat
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
NIA ==> Record(var:Symbol, fn:EDF, range:SOCDF, abserr:DF, relerr:DF)
INT ==> Integer
BOP ==> BasicOperator
URBODF ==> Union(Record(op:BasicOperator, w:DF), "failed")
LURBODF ==> List(Union(Record(op:BasicOperator, w:DF), "failed"))

E ==> with
  exprHasWeightCosWXorSinWX:NIA -> URBODF
  ++ \texttt{exprHasWeightCosWXorSinWX} looks for trigonometric
  ++ weights in an expression of the form \texttt{cos \omega x} or
  ++ \texttt{sin \omega x}, returning the value of \omega
  ++ (\texttt{notequal} 1) and the operator.
  exprHasAlgebraicWeight:NIA -> Union(LDF, "failed")
  ++ \texttt{exprHasAlgebraicWeight} looks for algebraic weights
  ++ giving rise to singularities of the function at the end-points.
  exprHasLogarithmicWeights:NIA -> INT
  ++ \texttt{exprHasLogarithmicWeights} looks for logarithmic weights
  ++ giving rise to singularities of the function at the end-points.

I ==> add
  score:(EDF, EDF) -> FI
  kernelIsLog:KEDF -> Boolean
  functionIsPolynomial?:EDF -> Boolean
  functionIsNthRoot?:(EDF, EDF) -> Boolean
  functionIsQuotient:EDF -> Union(EDF, "failed")
  findCommonFactor:LEDF -> Union(LEDF, "failed")
  findAlgebraicWeight:(NIA, EDF) -> Union(DF, "failed")
  exprHasListOfWeightsCosWXorSinWX:(EDF, Symbol) -> LURBODF
  exprOfFormCosWXorSinWX:(EDF, Symbol) -> URBODF
  bestWeight:LURBODF -> URBODF
  weightIn?:(URBODF, LURBODF) -> Boolean
  inRest?:(EDF, LEDF) -> Boolean
  factorIn?:(EDF, LEDF) -> Boolean
  voo?:(EDF, EDF) -> Boolean

  kernelIsLog(k:KEDF):Boolean ==
  (name k = (log :: Symbol))@Boolean
factorIn?(a: EDF, l: LEDF): Boolean ==
  for i in 1..# l repeat
    (a = l.i)@Boolean => return true
  false

voo?(b: EDF, a: EDF): Boolean ==
  (voo:=isTimes(b)) case LEDF and factorIn?(a, voo)

inRest?(a: EDF, l: LEDF): Boolean ==
  every?(x++voo?(x, a), l)

findCommonFactor(l: LEDF): Union(LEDF, "failed") ==
  empty?(l)$LEDF => "failed"
  f := first(l)$LEDF
  r := rest(l)$LEDF
  (t := isTimes(f)$EDF) case LEDF =>
    pos := select(x++inRest?(x, r), t)
    empty?(pos) => "failed"
    pos
  "failed"

exprIsLogarithmicWeight(f: EDF, Var: EDF, a: EDF, b: EDF): INT ==
  ans := 0$INT
  k := tower(f)$EDF
  lf := select(kernelIsLog, k)$LKEDF
  empty?(lf)$LKEDF => ans
  for i in 1..# lf repeat
    arg := argument lf.i
    if (arg.1 = (Var - a)) then
      ans := ans + 1
    else if (arg.1 = (b - Var)) then
      ans := ans + 2
  ans

exprHasLogarithmicWeights(args:NIA): INT ==
  ans := 1$INT
  a := getlo(args.range)$d01AgentsPackage :: EDF
  b := gethi(args.range)$d01AgentsPackage :: EDF
  Var := args.var :: EDF
  (l := isPlus numerator args.fn) case LEDF =>
    (cf := findCommonFactor l) case LEDF =>
      for j in 1..# cf repeat
        ans := ans + exprIsLogarithmicWeight(cf.j, Var, a, b)
      ans
  ans
  ans := ans + exprIsLogarithmicWeight(args.fn, Var, a, b)

functionIsQuotient(expr: EDF): Union(EDF, "failed") ==
  (k := mainKernel expr) case KEDF =>
    expr = inv(f := k :: KEDF :: EDF)$EDF => f
-- one?(numerator expr) => denominator expr
   (numerator expr = 1) => denominator expr
   "failed"
   "failed"

functionIsPolynomial?(f: EDF): Boolean ==
   (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

functionIsNthRoot?(f: EDF,e: EDF): Boolean ==
   (m := mainKernel f) case "failed" => false
   (one?((# (kernels f)))
   ((# (kernels f)) = 1)
   and (name operator m = (nthRoot :: Symbol))@Boolean
   and (((argument m).1 = e)@Boolean)

score(f: EDF,e: EDF): FI ==
   ans := 0$FI
   (t := isTimes f) case LEDF =>
      for i in 1..# t repeat
         ans := ans + score(t.i,e)
   ans
   (q := functionIsQuotient f) case EDF =>
      ans := ans - score(q,e)

functionIsPolynomial? f =>
   g: EDF := f/e
   if functionIsPolynomial? g then
      ans := 1+score(g,e)
   else
      ans
   (l := isPlus f) case LEDF =>
      (cf := findCommonFactor l) case LEDF =>
         factor := 1$EDF
         for i in 1..# cf repeat
            factor := factor*cf.i
         ans := ans + score(f/factor,e) + score(factor,e)
   ans

functionIsNthRoot?(f,e) =>
   (p := isPower f) case "failed" => ans
   exp := p.exponent
   m := mainKernel f
   m case KEDF =>
      arg := argument m
      a: INT := (retract(arg.2)@INT)$EDF
      exp / a
   ans

findAlgebraicWeight(args:NIA,e: EDF): Union(DF,"failed") ==
   zero?(s := score(args.fn,e)) => "failed"
   s :: DF
exprHasAlgebraicWeight(args:NIA):Union(LDF,"failed") ==
  (f := functionIsContinuousAtEndPoints(args)$d01AgentsPackage)
  case continuous =>"failed"

  Var := args.var :: EDF
  a := getlo(args.range)$d01AgentsPackage :: EDF
  b := gethi(args.range)$d01AgentsPackage :: EDF
  A := Var - a
  B := b - Var
  f case lowerSingular =>
    (h := findAlgebraicWeight(args,A)) case "failed" => "failed"
    [h,0]
  f case upperSingular =>
    (g := findAlgebraicWeight(args,B)) case "failed" => "failed"
    [0,g]
  h := findAlgebraicWeight(args,A)
  g := findAlgebraicWeight(args,B)
  r := (h case "failed")
  s := (g case "failed")
  (r) and (s) => "failed"
  r => [0,coerce(g)@DF]
  s => [coerce(h)@DF,0]
  [coerce(h)@DF,coerce(g)@DF]

exprOfFormCosWXorSinWX(f:EDF,var:Symbol): URBODF ==
  l:LKEDF := kernels(f)$EDF
  -- one?((# l)$LKEDF)$INT =>
  # l = 1 =>
    a:LEDF := argument(e:KEDF := first(l)$LKEDF)$KEDF
    empty?(a) => "failed"
    m:Union(LEDF,"failed") := isTimes(first(a)$LEDF)$EDF
    m case LEDF => -- if it is a list, it will have at least two elements
      is?(second(m)$LEDF,var)@EDF =>
        omega:DF := retract(first(m)$LEDF)@DF
        o:BOP := operator(n:Symbol:=name(e)$KEDF)$BOP
        (n = cos@Symbol)@Boolean => [o,omega]
        (n = sin@Symbol)@Boolean => [o,omega]
        "failed"
        "failed"
        "failed"
        "failed"
      "failed"
      "failed"
      "failed"

exprHasListOfWeightsCosWXorSinWX(f:EDF,var:Symbol): LURBODF ==
  (e := isTimes(f)$EDF) case LEDF =>
    [exprOfFormCosWXorSinWX(u,var) for u in e]
  empty?(k := kernels f) => ["failed"]
  (first(k):EDF) = f =>
    [exprOfFormCosWXorSinWX(f,var)]
    ["failed"]
bestWeight(l:LURBODF): URBODF ==
  empty?(l)$LURBODF => "failed"
  best := first(l)$LURBODF -- best is first in list
  empty?(rest(l)$LURBODF) => best
  for i in 2..# l repeat -- unless next is better
    r:URBODF := l.i
    if r case "failed" then leave
    else if best case "failed" then
      best := r
    else if r.w > best.w then
      best := r
  best

weightIn?(weight:URBODF,listOfWeights:LURBODF):Boolean ==
  n := # listOfWeights
  for i in 1..n repeat -- cycle through list
    (weight = listOfWeights.i)$Boolean => return true -- return when found
  false

exprHasWeightCosWXorSinWX(args:NIA):URBODF ==
  ans := empty()$LURBODF
  f:EDF := numerator(args.fn)$EDF
  (t:Union(LEDF,"failed") := isPlus(f)) case "failed" =>
    bestWeight(exprHasListOfWeightsCosWXorSinWX(f,args.var))
  if t case LEDF then
    e1 := first(t)$LEDF
    le1:LURBODF := exprHasListOfWeightsCosWXorSinWX(e1,args.var)
    le1 := [u for u in le1 | (not (u case "failed"))] [1]
    empty?(le1)$LURBODF => "failed"
    test := true
    for i in 1..# le1 repeat
      le1i:URBODF := le1.i
      for j in 2..# t repeat
        if test then
          tj:LEDBODF := exprHasListOfWeightsCosWXorSinWX(t.j,args.var)
          test := weightIn?(le1i,tj)
        if test then
          ans := concat([le1i],ans)
        bestWeight ans
      else "failed"
  "failed"

—— D01WGTS.dotabb ——

"D01WGTS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D01WGTS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D01WGTS" -> "FS"
package D02AGNT d02AgentsPackage

---

| d02AgentsPackage.input |
---

)set break resume
)sys rm -f d02AgentsPackage.output
)spool d02AgentsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d02AgentsPackage
--E 1

)spool
)lisp (bye)

---

| d02AgentsPackage.help |
---

====================================================================
d02AgentsPackage examples
====================================================================

D02AgentsPackage contains a set of computational agents for use with Ordinary Differential Equation solvers.

See Also:
o )show d02AgentsPackage
d02AgentsPackage (D02AGNT)

Exports:
accuracyIF combineFeatureCompatibility eval expenseOfEvaluationIF
jacobian sparsityIF stiffnessAndStabilityFactor stiffnessAndStabilityOfODE

— package D02AGNT d02AgentsPackage —

)abbrev package D02AGNT d02AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: January 1997
++ Description:

d02AgentsPackage(): E == I where
  LEDF ==> List Expression DoubleFloat
  LEEDF ==> List Equation Expression DoubleFloat
  EEDF ==> Equation Expression DoubleFloat
  VEDF ==> Vector Expression DoubleFloat
  MEDF ==> Matrix Expression DoubleFloat
  MDF ==> Matrix DoubleFloat
  EDF ==> Expression DoubleFloat
  DF ==> DoubleFloat
  F ==> Float
  INT ==> Integer
  CDF ==> Complex DoubleFloat
  LDF ==> List DoubleFloat
  LF ==> List Float
  S ==> Symbol
  LS ==> List Symbol
  MFI ==> Matrix Fraction Integer
  LFI ==> List Fraction Integer
  FI ==> Fraction Integer
  ON ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
  RVE ==> Record(val:EDF,exponent:INT)
  RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
ROA ==> Record(key:ODEA, entry:ATT)
E ==> with
  combineFeatureCompatibility: (F,F) -> F
  ++ combineFeatureCompatibility(C1,C2) is for interacting attributes
  combineFeatureCompatibility: (F,LF) -> F
  ++ combineFeatureCompatibility(C1,L) is for interacting attributes
  sparsityIF: MEDF -> F
  ++ sparsityIF(m) calculates the sparsity of a jacobian matrix
  jacobian: (VEDF,LS) -> MEDF
  ++ jacobian(v,w) is a local function to make a jacobian matrix
  eval: (MEDF,LS,VEDF) -> MEDF
  ++ eval(mat,symbols,values) evaluates a multivariable matrix at given values
  ++ for each of a list of variables
  stiffnessAndStabilityFactor: MEDF -> RSS
  ++ stiffnessAndStabilityFactor(me) calculates the stability and
  ++ stiffness factor of a system of first-order differential equations
  ++ (by evaluating the maximum difference in the real parts of the
  ++ negative eigenvalues of the jacobian of the system for which O(10)
  ++ equates to mildly stiff whereas stiffness ratios of O(10^6) are not
  ++ uncommon) and whether the system is likely to show any oscillations
  ++ (identified by the closeness to the imaginary axis of the complex
  ++ eigenvalues of the jacobian).
  stiffnessAndStabilityOfODEIF:ODEA -> RSS
  ++ stiffnessAndStabilityOfODEIF(ode) calculates the intensity values
  ++ of stiffness of a system of first-order differential equations
  ++ (by evaluating the maximum difference in the real parts of the
  ++ negative eigenvalues of the jacobian of the system for which O(10)
  ++ equates to mildly stiff whereas stiffness ratios of O(10^6) are not
  ++ uncommon) and whether the system is likely to show any oscillations
  ++ (identified by the closeness to the imaginary axis of the complex
  ++ eigenvalues of the jacobian).
  ++
  ++ It returns two values in the range [0,1].
  systemSizeIF:ODEA -> F
  ++ systemSizeIF(ode) returns the intensity value of the size of
  ++ the system of ODEs. 20 equations corresponds to the neutral
  ++ value. It returns a value in the range [0,1].
  expenseOFEvaluationIF:ODEA -> F
  ++ expenseOFEvaluationIF(o) returns the intensity value of the
  ++ cost of evaluating the input ODE. This is in terms of the number
  ++ of ‘‘operational units’’ . It returns a value in the range
  ++ [0,1].
  ++ 400 ‘‘operation units’’ -> 0.75
  ++ 200 ‘‘operation units’’ -> 0.5
  ++ 83 ‘‘operation units’’ -> 0.25
  ++ exponentiation = 4 units , function calls = 10 units.
  accuracyIF:ODEA -> F
  ++ accuracyIF(o) returns the intensity value of the accuracy
++ requirements of the input ODE. A request of accuracy of $10^{-6}$
++ corresponds to the neutral intensity. It returns a value
++ in the range $[0,1]$.

intermediateResultsIF: ODEA -> F
++ intermediateResultsIF(ode) returns a value corresponding to the
++ required number of intermediate results required and, therefore,
++ an indication of how much this would affect the step-length of the
++ calculation. It returns a value in the range $[0,1]$.

I ==> add

import ExpertSystemToolsPackage
accuracyFactor: ODEA -> F
expenseOfEvaluation: ODEA -> F
eval1: (LEDF, LLEDF) -> LEDF
stiffnessAndStabilityOfODE: ODEA -> RSS
intermediateResultsFactor: ODEA -> F
leastStabilityAngle: (LDF, LDF) -> F

intermediateResultsFactor(ode: ODEA): F ==
resultsRequirement := #(ode.intvals)
(1.0-exp(-(resultsRequirement::F)/50.0)$F)

intermediateResultsIF(ode: ODEA): F ==

ode := copy ode
(t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
s := coerce(t)@ATT
negative?(s.intermediateResults)$F =>
s.intermediateResults := intermediateResultsFactor(ode)
r:ROA := [ode, s]
insert!(r)$ODEIntensityFunctionsTable
s.intermediateResults

a: ATT := [-1.0, -1.0, -1.0, -1.0, e := intermediateResultsFactor(ode)]
r: ROA := [ode, a]
insert!(r)$ODEIntensityFunctionsTable
e

accuracyFactor(ode: ODEA): F ==
accuracyRequirements := convert(ode.abserr)$F
if zero?(accuracyRequirements) then
accuracyRequirements := convert(ode.relerr)$F
val := inv(accuracyRequirements)$F
n := log10(val)$F
(1.0-exp(-(n/(2.0))**2/(15.0))$F)

accuracyIF(ode: ODEA): F ==

ode := copy ode
(t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
s := coerce(t)@ATT
negative?(s.accuracy)$F =>
s.accuracy := accuracyFactor(ode)
r:ROA := [ode,s]
insert!(r)$ODEIntensityFunctionsTable
s.accuracy
s.accuracy

a:ATT := [-1.0,-1.0,-1.0,e:=accuracyFactor(ode),-1.0]
r:ROA := [ode,a]
insert!(r)$ODEIntensityFunctionsTable

systemSizeIF(ode:ODEA):F ==
n := #(ode.fn)
(1.0-exp((-n::F/75.0))$F)

expenseOfEvaluation(o:ODEA):F ==
-- expense of evaluation of an ODE -- <0.3 inexpensive - 0.5 neutral - >0.7 very expensive
-- 400 'operation units' → 0.75
-- 200 'operation units' → 0.5
-- 83 'operation units' → 0.25
-- ** = 4 units , function calls = 10 units.
oke := copy o.fn
expenseOfEvaluation(ode)

expenseOfEvaluationIF(o:ODEA):F ==
oke := copy o
(t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
s := coerce(t)@ATT
negative?(s.expense)$F =>
s.expense := expenseOfEvaluation(ode)
r:ROA := [ode,s]
insert!(r)$ODEIntensityFunctionsTable
s.expense
s.expense

a:ATT := [-1.0,-1.0,e:=expenseOfEvaluation(ode),-1.0,-1.0]
r:ROA := [ode,a]
insert!(r)$ODEIntensityFunctionsTable

leastStabilityAngle(realPartsList:LDF,imagPartsList:LDF):F ==
complexList := [complex(u,v)$CDF for u in realPartsList for v in imagPartsList]
argumentList := [abs((abs(argument(u)$CDF)$DF)-(pi()$DF)/2)$DF for u in complexList]
sortedArgumentList := sort(argumentList)$LDF
list := [u for u in sortedArgumentList | not zero?(u) ]
empty?(list)$LDF => 0$F
convert(first(list)$LDF)$F

stiffnessAndStabilityFactor(me:MEDF):RSS ==
-- search first for real eigenvalues of the jacobian (symbolically)
-- if the system isn’t too big
r:INT := ncols(me)$MEDF
b:Boolean := ((# me) < 150)
if b then
  mc:MFI := map(edf2fi,me)$ExpertSystemToolsPackage2(EDF,FI)
e:LFI := realEigenvalues(mc,1/100)$NumericRealEigenPackage(FI)
b := ((# e) >= r-1)@Boolean
if b then
  -- if all the eigenvalues are real, find negative ones
  e := sort(neglist(e)$ExpertSystemToolsPackage1(DF))
  -- if there are two or more, calculate stiffness ratio
  ((n:=#e)>1)@Boolean => [coerce(e.1/e.n)@F,0$F]
  -- otherwise stiffness not present
  [0$F,0$F]
else

  md:MDF := map(edf2df,me)$ExpertSystemToolsPackage2(EDF,DF)

  -- otherwise calculate numerically the complex eigenvalues
  -- using NAG routine f02aff.

  res:Result := f02aff(r,r,md,-1)$NagEigenPackage

  realParts:Union(Any,"failed") := search(rr::Symbol,res)$Result
  realParts case "failed" => [0$F,0$F]
  realPartsMatrix:MDF := retract(realParts)$AnyFunctions1(MDF) -- array === matrix
  imagParts:Union(Any,"failed") := search(ri::Symbol,res)$Result
  imagParts case "failed" => [0$F,0$F]
  imagPartsMatrix:MDF := retract(imagParts)$AnyFunctions1(MDF) -- array === matrix
  imagPartsList:LDF := members(imagPartsMatrix)$MDF
  realPartsList:LDF := members(realPartsMatrix)$MDF
  stabilityAngle := leastStabilityAngle(realPartsList,imagPartsList)
  negRealPartsList := sort(neglist(realPartsList)$ExpertSystemToolsPackage1(DF))
  empty?(negRealPartsList)$LDF => 
  out := convert(negRealPartsList.1/negRealPartsList.n)@F
  stabilityAngle
  [out,0$F]
  [out,0$F]
  [out,0$F]
  [out,0$F]
  [out,0$F]
  [out,0$F]

  eval1(l:LEDF,e:LEEDF):LEDF ==
  [eval(u,e)$EDF for u in l]

  eval(mat:MEDF,symbols:LS,values:VEDF):MEDF ==
  l := listOfLists(mat)
  ledf := entries(values)$VEDF
  e := [equation(u::EDF,v)$EEDF for u in symbols for v in ledf]
  l := [eval1(w,e) for w in l]
  matrix l

  combineFeatureCompatibility(C1:F,C2:F):F ==
-- C1 C2
-- s(C1,C2) = -----------------------
-- C1 C2 + (1 - C1)(1 - C2)

C1*C2/((C1*C2)+(1$F-C1)*(1$F-C2))

combineFeatureCompatibility(C1:F,L:LF):F ==

empty?(L)$LF => C1
C2 := combineFeatureCompatibility(C1,first(L)$LF)
    combineFeatureCompatibility(C2,rest(L)$LF)

jacobian(v:VEDF,w:LS):Matrix EDF ==
    jacobian(v,w)$MultiVariableCalculusFunctions(S,EDF,VEDF,LS)

sparsityIF(m:Matrix EDF):F ==
    l:LEDF :=parts m
    z:LEDF := [u for u in l | zero?(u)$EDF]
    ((#z)::F/(#l)::F)

sum(a:EDF,b:EDF):EDF == a+b

stiffnessAndStabilityOfODE(ode:ODEA):RSS ==
    odefs := copy ode.fn
    ls:LS := [subscript(Y,[coerce(n)])$Symbol for n in 1..# odefs]
    yvals := copy ode.yinit
    for i in 1..#yvals repeat
        zero?(yvals.i) => yvals.i := 0.1::DF
    yeexpr := [coerce(v)@EDF for v in yvals]
    yv:VEDF := vector(yexpr)
    j1:MEDF := jacobian(odefn,ls)
    ej1:MEDF := eval(j1,ls,yv)
    ej1 := eval(ej1,variables(reduce(sum,members(ej1)$MEDF)),vector([ode.xinit::EDF]))
    ssf := stiffnessAndStabilityFactor(ej1)
    stability := 1.0-sqrt((ssf.stabilityFactor)*(2.0)/(pi()$F))
    stiffness := (1.0)-exp(-(ssf.stiffnessFactor)/(500.0))
    [stiffness,stability]

stiffnessAndStabilityOfODEIF(ode:ODEA):RSS ==
    odefn := copy ode
    (t := showIntensityFunctions(odefn)$ODEIntensityFunctionsTable) case ATT =>
    s:ATT := coerce(t)@ATT
    negative?(s.stiffness)$F =>
        ssf:RSS := stiffnessAndStabilityOfODE(odefn)
        s := [ssf.stiffnessFactor,ssf.stabilityFactor,s.expense,
                s.accuracy,s.intermediateResults]
    r:ROA := [odefn,s]
        insert!(r)$ODEIntensityFunctionsTable
        ssf
    [s.stiffness,s.stability]
ssf:RSS := stiffnessAndStabilityOfODE(odefn)
s:ATT := [ssf.stiffnessFactor,ssf.stabilityFactor,-1.0,-1.0,-1.0]
r:ROA := [odefn,s]
insert!(r)$ODEIntensityFunctionsTable
ssf

---

[D2AGNT.dotabb](#)

"D02AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D02AGNT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D02AGNT" -> "FS"

---

package D03AGNT d03AgentsPackage

--- d03AgentsPackage.input ---

)set break resume
)sys rm -f d03AgentsPackage.output
)spool d03AgentsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d03AgentsPackage
--E 1

)spool
)lisp (bye)

---

--- d03AgentsPackage.help ---

====================================================================
d03AgentsPackage examples
====================================================================

d03AgentsPackage contains a set of computational agents
for use with Partial Differential Equation solvers.

See Also:
- )show d03AgentsPackage

---

d03AgentsPackage (D03AGNT)

Exports:
- central?
- elliptic?
- subscriptedVariables
- varList

— package D03AGNT d03AgentsPackage —

)abbrev package D03AGNT d03AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: December 1997
++ Description:
++ \texttt{d03AgentsPackage} contains a set of computational agents
++ for use with Partial Differential Equation solvers.

d03AgentsPackage(): E == I where
  LEDF ==> List Expression DoubleFloat
  EDF ==> Expression DoubleFloat
  MDF ==> Matrix DoubleFloat
  DF ==> DoubleFloat
  F ==> Float
  INT ==> Integer
  NNI ==> NonNegativeInteger
  EEDF ==> Equation Expression DoubleFloat
  LEEDF ==> List Equation Expression DoubleFloat
  LDF ==> List DoubleFloat
LOCDF ==> List OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
LS ==> List Symbol
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
    dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
    f:List LEDF, st:String, tol:DF)
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)

E ==> with
    varList:(Symbol,NonNegativeInteger) -> LS
    ++ varList(s,n) \undocumented{}
    subscriptedVariables:EDF -> EDF
    ++ subscriptedVariables(e) \undocumented{}
    central?:(DF,DF,LEDF) -> Boolean
    ++ central?(f,g,l) \undocumented{}
    elliptic?:PDEB -> Boolean
    ++ elliptic?(r) \undocumented{}

I ==> add

import ExpertSystemToolsPackage

sum(a:EDF,b:EDF):EDF == a+b
varList(s:Symbol,n:NonNegativeInteger):LS ==
    [subscript(s,[t::OutputForm]) for t in expand([1..n])$Segment(Integer)]

subscriptedVariables(e:EDF):EDF ==
    oldVars:List Symbol := variables(e)
    o := [a :: EDF for a in oldVars]
    newVars := varList(X::Symbol,# oldVars)
    n := [b :: EDF for b in newVars]
    subst(e, [a=b for a in o for b in n])

central?(x:DF,y:DF,p:LEDF):Boolean ==
    ls := variables(reduce(sum,p))
    le := [equation(u::EDF,v)$EDF for u in ls for v in [x::EDF,y::EDF]]
    l := [eval(u,le)$EDF for u in p]
    max(1.4,1.5) < 20 * max(1.1,max(1.2,1.3))

elliptic?(args:PDEB):Boolean ==
    (args.st)="elliptic" => true
    p := args.pde
    xcon:PDEC := first(args.constraints)
    ycon:PDEC := second(args.constraints)
    xs := xcon.start
    ys := ycon.start
    xf := xcon.finish
    yf := ycon.finish
xstart:DF := (xf-xs)/2$DF
ystart:DF := (yf-ys)/2$DF
optStart:LDF := [xstart,ystart]
lower:LOCDF := [xs::OCDF,ys::OCDF]
upper:LOCDF := [xf::OCDF,yf::OCDF]
v := variables(e := 4*first(p)*third(p)-(second(p))**2)
eq := subscriptedVariables(e)
noa:NOA :=
  -- one?(# v) =>
  (# v) = 1 =>
    ((first v) = X$Symbol) =>
    [eq,[xstart],[xs::OCDF],empty()$LED,,[xf::OCDF]]
    [eq,[ystart],[ys::OCDF],empty()$LED,,[yf::OCDF]]
    [eq,lower,empty()$LED,upper]
ell := optimize(noa::NumericalOptimizationProblem)$AnnaNumericalOptimizationPackage
o:Union(Any,"failed") := search(objf::Symbol,ell)$Result
o case "failed" => false
ob := o :: Any
obj:DF := retract(ob)$AnyFunctions1(DF)
positive?(obj)
Chapter 6

Chapter E

package EP EigenPackage

— EigenPackage.input —

)set break resume
)sys rm -f EigenPackage.output
)spool EigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EigenPackage
--E 1

)spool
)lisp (bye)

———

— EigenPackage.help —

====================================================================
EigenPackage examples
====================================================================

This is a package for the exact computation of eigenvalues and eigenvectors. This package can be made to work for matrices with coefficients which are rational functions over a ring where we can factor polynomials. Rational eigenvalues are always explicitly computed while the
non-rational ones are expressed in terms of their minimal polynomial.

See Also:
• )show EigenPackage

EigenPackage (EP)

Exports:
• characteristicPolynomial
• eigenvalues
• eigenvector
• eigenvectors
• generalizedEigenvector
• generalizedEigenvectors

— package EP EigenPackage —

)abbrev package EP EigenPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Date Last Updated: October 1992
++ Description:
++ This is a package for the exact computation of eigenvalues and eigenvectors.
++ This package can be made to work for matrices with coefficients which are
++ rational functions over a ring where we can factor polynomials.
++ Rational eigenvalues are always explicitly computed while the
++ non-rational ones are expressed in terms of their minimal
++ polynomial.
-- Functions for the numeric computation of eigenvalues and eigenvectors
-- are in numeigen.spad.

EigenPackage(R) : C == T
where
R : GcdDomain
P ==> Polynomial R
F ==> Fraction P
SE ==> Symbol()
SUP ==> SparseUnivariatePolynomial(P)
SUF ==> SparseUnivariatePolynomial(F)
M ==> Matrix(F)
NNI ==> NonNegativeInteger
ST ==> SuchThat(SE,P)

Eigenvalue ==> Union(F,ST)
EigenForm ==> Record(eigval:Eigenvalue,eigmult:NNI,eigvec : List M)
GenEigen ==> Record(eigval:Eigenvalue,geneigvec:List M)

C == with
  characteristicPolynomial : (M,Symbol) -> P
  ++ characteristicPolynomial(m,var) returns the
  ++ characteristicPolynomial of the matrix m using
  ++ the symbol var as the main variable.

  characteristicPolynomial : M -> P
  ++ characteristicPolynomial(m) returns the
  ++ characteristicPolynomial of the matrix m using
  ++ a new generated symbol symbol as the main variable.

  eigenvalues : M -> List Eigenvalue
  ++ eigenvalues(m) returns the
  ++ eigenvalues of the matrix m which are expressible
  ++ as rational functions over the rational numbers.

  eigenvector : (Eigenvalue,M) -> List M
  ++ eigenvector(eigval,m) returns the
  ++ eigenvectors belonging to the eigenvalue eigval
  ++ for the matrix m.

  generalizedEigenvector : (Eigenvalue,M,NNI,NNI) -> List M
  ++ generalizedEigenvector(alpha,m,k,g) returns the generalized eigenvectors
  ++ of the matrix relative to the eigenvalue alpha.
  ++ The integers k and g are respectively the algebraic and the
  ++ geometric multiplicity of the eigenvalue alpha.
  ++ alpha can be either rational or not.
  ++ in the second case alpha is the minimal polynomial of the
  ++ eigenvalue.

  generalizedEigenvector : (EigenForm,M) -> List M
  ++ generalizedEigenvector(eigen,m) returns the generalized eigenvectors
  ++ of the matrix relative to the eigenvalue eigen, as
  ++ returned by the function eigenvectors.

  generalizedEigenvectors : M -> List GenEigen
+ generalizedEigenvectors(m)  
+ returns the generalized eigenvectors  
+ of the matrix m.

eigenvectors : M -> List(EigenForm)  
+ eigenvectors(m) returns the eigenvalues and eigenvectors  
+ for the matrix m.  
+ The rational eigenvalues and the correspondent eigenvectors  
+ are explicitly computed, while the non rational ones  
+ are given via their minimal polynomial and the corresponding  
+ eigenvectors are expressed in terms of a "generic" root of  
+ such a polynomial.

T == add
PI     ==> PositiveInteger

MF     := GeneralizedMultivariateFactorize(SE, IndexedExponents SE, R, R, F)
UPCF2  := UnivariatePolynomialCategoryFunctions2(P, SUP, F, SUF)

---- Local Functions ----
tff     : (SUF, SE) -> F
fft     : (SUF, SE) -> F
charpol : (M, SE) -> F
intRatEig : (F, M, NNI) -> List M
intAlgEig : (ST, M, NNI) -> List M
genEigForm : (EigenForm, M) -> GenEigen

---- next functions needed for defining ModularField ----
reduction(u:SUF, p:SUF):SUF == u rem p
merge(p:SUF, q:SUF):Union(SUF, "failed") ==
    p = q => p
    p = 0 => q
    q = 0 => p
    "failed"
exactquo(u:SUF, v:SUF, p:SUF):Union(SUF, "failed") ==
    val == extendedEuclidean(v, p, u)
    val case "failed" => "failed"
    val.coef1

---- functions for conversions ----
fft(t:SUF, x:SE) : F ==
    n = degree(t)
    cf == monomial(1, x, n) $ P :: F
    cf * leadingCoefficient t

tff(p:SUF, x:SE) : F ==
PACKAGE EP EIGENPACKAGE

degree p=0 => leadingCoefficient p
r:F:=0$F
while p=true repeat
    r:=r+fft(p,x)
p := reductum p
r

---- generalized eigenvectors associated to a given eigenvalue ----

genEigForm(eigen : EigenForm,A:M) : GenEigen ==
    alpha:=eigen.eigval
    k:=eigen.eigmult
    g:=(eigen.eigvec)
    k = g => [alpha,eigen.eigvec]
    [alpha,generalizedEigenvector(alpha,A,k,g)]

---- characteristic polynomial ----

ccharpol(A:M,x:SE) : F ==
    dimA :PI := (nrows A):PI
    dimA ^= ncols A => error " The matrix is not square"
    B:M:=zero(dimA,dimA)
    for i in 1..dimA repeat
        for j in 1..dimA repeat B(i,j):=A(i,j)
        B(i,i) := B(i,i) - monomial(1$P,x,1)::F
    determinant B

-------- EXPORTED FUNCTIONS --------

---- characteristic polynomial of a matrix A ----

ccharpol(A:M):P ==
    x:SE:=new()$SE
    numer ccharpol(A,x)

---- characteristic polynomial of a matrix A ----

ccharpol(A:M,x:SE) : P == numer ccharpol(A,x)

---- Eigenvalues of the matrix A ----

eigenvalues(A:M): List Eigenvalue ==
    x:=new()$SE
    pol:= ccharpol(A,x)
    lrat:List F :=empty()
    lsym:List ST :=empty()
    for eq in solve(pol,x)$SystemSolvePackage(R) repeat
        alg:=numer lhs eq
        degree(alg, x)=1 => lrat:=cons(rhs eq,lrat)
        lsym:=cons([x,alg],lsym)
    append([lr::Eigenvalue for lr in lrat],
            [ls::Eigenvalue for ls in lsym])

---- Eigenvectors belonging to a given eigenvalue ----
---- the eigenvalue must be exact ----
eigenvector(alpha:Eigenvalue,A:M) : List M ==
alpha case F => intRatEig(alpha::F,A,1$NNI)
intAlgEig(alpha::ST,A,1$NNI)

----- Eigenvectors belonging to a given rational eigenvalue ----- 
----- Internal function ----- 
intRatEig(alpha:F,A:M,m:NNI) : List M ==
n:=nrows A
B:M := zero(n,n)$M
for i in 1..n repeat
  for j in 1..n repeat B(i,j):=A(i,j)
  B(i,i):= B(i,i) - alpha
[v::M for v in nullSpace(B**m)]

----- Eigenvectors belonging to a given algebraic eigenvalue ----- 
------ Internal Function ------ 
intAlgEig(alpha:ST,A:M,m:NNI) : List M ==
n:=nrows A
MM := ModularField(SUF,SUF,reduction,merge,exactquo)
AM:=Matrix MM
x:SE:=lhs alpha
pol:SUF:=unitCanonical map(coerce,univariate(rhs alpha,x))$UPCF2
alg:MM:=reduce(monomial(1,1),pol)
B:AM := zero(n,n)
for i in 1..n repeat
  for j in 1..n repeat B(i,j):=reduce(A(i,j)::SUF,pol)
  B(i,i):= B(i,i) - alg
sol: List M :=empty()
for vec in nullSpace(B**m) repeat
  w:M:=zero(n,1)
  for i in 1..n repeat w(i,1):=tff((vec.i)::SUF,x)
sol:=cons(w,sol)
sol

----- Generalized Eigenvectors belonging to a given eigenvalue ---- 
generalizedEigenvector(alpha:Eigenvalue,A:M,k:NNI,g:NNI) : List M ==
alpha case F => intRatEig(alpha::F,A,(1+k-g)::NNI)
intAlgEig(alpha::ST,A,(1+k-g)::NNI)

----- Generalized Eigenvectors belonging to a given eigenvalue ----
generalizedEigenvector(eigen :EigenForm,A:M) : List M ==
generalizedEigenvector(eigen.eigval,A,eigen.eigmult,# eigen.eigvec)

----- Generalized Eigenvectors ----- 
generalizedEigenvectors(A:M) : List GenEigen ==
n:=nrows A
leig:=eigenvectors A
[genEigForm(leg,A) for leg in leig]

----- eigenvectors and eigenvalues -----
eigenvectors(A:M):List(EigenForm) ==
n:=nrows A
x:=new()$SE
p:=numer charpol(A,x)
MM := ModularField(SUF,SUF,reduction,merge,exactquo)
AM:=Matrix(MM)
ratSol : List EigenForm := empty()
algSol : List EigenForm := empty()
lff:=factors factor p
for fact in lff repeat
  pol:=fact.factor
  degree(pol,x)=1 =>
    vec:F :=-coefficient(pol,x,0)/coefficient(pol,x,degree(pol,x))
    ratSol:=cons([vec,fact.exponent :: NNI, intRatEig(vec,A,1$NNI)]$EigenForm,ratSol)
  alpha:ST:=[x,pol]
  algSol:=cons([alpha,fact.exponent :: NNI, intAlgEig(alpha,A,1$NNI)]$EigenForm,algSol)
append(ratSol,algSol)

— EP.dotabb —
"EP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"EP" -> "ALIST"

package EF ElementaryFunction

— ElementaryFunction.input —

)set break resume
)sys rm -f ElementaryFunction.output
)spool ElementaryFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 32
)trace EF
--R
Parameterized constructors traced:

\[ D(\cos(3x + 6y), x) \]

\[ (1) - 3\sin(6y + 3x) \]

Type: Expression(Integer)
Note that both Mathematica and Maxima return this result as $3\sec(3x + 6y)^2$ but Maple returns the same result as Axiom. They are equivalent results.

Note that Mathematica and Maxima return $-3\csc(3x + 6y)^2$ and Maple returns the same form as Axiom. They are equivalent.
simplify \((-3\cot(6y+3x)^2+3) -(-3\csc(3x+6y)^2)\)

\[\text{(7) 0}\]

Type: Expression(Integer)

\(\text{D(sec}(3x+6y),x)\)

\[\text{(8) 3sec}(6y + 3x)\tan(6y + 3x)\]

Type: Expression(Integer)

\(\text{D(csc}(3x+6y),x)\)

\[\text{(9) - 3cot}(6y + 3x)\csc(6y + 3x)\]

Type: Expression(Integer)

\(\text{D(asin}(3x+6y),x)\)

\[\text{(10) \frac{1}{2 - 36y - 36x y - 9x + 1}}\]

Type: Expression(Integer)

\(\text{D(acos}(3x+6y),x)\)

\[\text{(11) \frac{1}{2 - 36y - 36x y - 9x + 1}}\]

Type: Expression(Integer)
\[ D(\tan(3x+6y),x) \]

\[ 3 \]

\[ \frac{2}{2} \]

\[ 36y + 36x y + 9x + 1 \]

Type: Expression(Integer)

\[ D(\cot(3x+6y),x) \]

\[ 3 \]

\[ \frac{2}{2} \]

\[ 36y + 36x y + 9x + 1 \]

Type: Expression(Integer)

Mathematica, Maple, and Maxima give:

\[ \frac{3}{(3x+6y)^2} \sqrt{1 - \frac{1}{(3x+6y)^2}} \]

which proceeds directly from the formula for the derivative of \( \text{arcsec}(x) \):

\[ \frac{d}{dx} \text{arcsec}(x) = \frac{1}{x\sqrt{x^2 - 1}} \]

---

\[ D(\text{asec}(3x+6y),x) \]

\[ 1 \]

\[ \frac{1}{2} \]

\[ (2y + x)\sqrt{36y + 36x y + 9x - 1} \]

Type: Expression(Integer)
If we use the same formula for this example:

\[
\frac{1}{(3x + 6y)\sqrt{(3x + 6y)^2 - 1}} \, d(3x + 6y)/dx
\]

--- ElementaryFunction.input ---

-- S 17 of 32

\[
\frac{3}{((3x + 6y)\sqrt{(3x + 6y)^2 - 1})}
\]

-- R

-- R

-- R (15) \[ \frac{1}{(2y + x)\sqrt{36y + 36x y + 9x - 1}} \]

-- R Type: Expression(Integer)

---

Mathematica, Maple, and Maxima give

\[
-\frac{3}{(3x + 6y)^2\sqrt{1 - \frac{1}{(3x + 6y)^2}}}
\]

which is just the negative of the above result and we can see that the same analysis applies to explain the results.

--- ElementaryFunction.input ---

-- S 18 of 32

\[
D(\text{acsc}(3x + 6y), x)
\]

-- R

-- R

-- R (16) \[ \frac{1}{(2y + x)\sqrt{36y + 36x y + 9x - 1}} \]

-- R Type: Expression(Integer)

---

-- S 19 of 32

\[
D(\text{sinh}(3x + 6y), x)
\]

-- R

-- R (17) \[ 3\text{cosh}(6y + 3x) \]
Mathematica and Maxima return

\[3\text{sech}(3x + 6y)^2\]

Maple returns Axiom’s answer. Both are equivalent.

—— ElementaryFunction.input ——

Mathematica and Maxima return

\[-3\text{csch}(3x + 6y)^2\]

Maple returns Axiom’s answer. Both are equivalent.

—— ElementaryFunction.input ——
Mathematica and Maple show

\[
\frac{3}{\sqrt{-1 + 3x + 6y} \sqrt{1 + 3x + 6y}}
\]

Maxima gives Axiom’s answer. Both are equivalent, just factored forms.
--- ElementaryFunction.input ---

--S 28 of 32
D(acosh(3*x+6*y),x)
--R
--R
--R 3
--R (26) ------------------------+
--R | 2 2
--R \|36y + 36x y + 9x - 1
--R Type: Expression(Integer)
--E 28

--S 29 of 32
D(atanh(3*x+6*y),x)
--R
--R
--R 3
--R (27) - ----------------------
--R 2 2
--R 36y + 36x y + 9x - 1
--R Type: Expression(Integer)
--E 29

--S 30 of 32
D(acoth(3*x+6*y),x)
--R
--R
--R 3
--R (28) - ----------------------
--R 2 2
--R 36y + 36x y + 9x - 1
--R Type: Expression(Integer)
--E 30

__________

Mathematica gives
\[-\frac{3}{(3x + 6y)\sqrt{\frac{1-3x-6y}{13x+6y}(1 + 3x + 6y)}}\]

Maxima gives
\[-\frac{3}{(6y + 3x)^2 \sqrt{\frac{1}{(6y+3x)^2} - 1}}\]

Maple gives
\[-\frac{3}{(3x + 6y)^2 \sqrt{\frac{1}{3x+6y} - 1} \sqrt{\frac{1}{3x+6y} + 1}}\]
Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.

--- ElementaryFunction.input ---

```plaintext
--S 31 of 32
D(asech(3*x+6*y),x)
--R
--R
--R
--R (29)  - -------------------------------
--R +------------------------+
--R | 2 2
--R (2y + x)| 36y - 36x y - 9x + 1
--R Type: Expression(Integer)
--E 31
```

Mathematica, Maple, and Maxima all generate the answer

\[
-\frac{3}{(3x + 6y)^2 \sqrt{1 + \frac{1}{(3x+6y)^2}}}
\]

Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.

--- ElementaryFunction.input ---

```plaintext
--S 32 of 32
D(acsch(3*x+6*y),x)
--R
--R
--R
--R (30)  - -------------------------------
--R +----------------------+
--R | 2 2
--R (2y + x)\|36y + 36x y + 9x + 1
--R Type: Expression(Integer)
--E 32
```

)spool
)lisp (bye)

---

--- ElementaryFunction.help ---

====================================================================
This package provides elementary functions over an integral domain.

\[ D(\sin(3x+6y), x) \]
\[ = 3\cos(6y + 3x) \]

\[ D(\cos(3x+6y), x) \]
\[ = -3\sin(6y + 3x) \]

\[ D(\tan(3x+6y), x) \]
\[ = 2 \cdot 3\tan(6y + 3x) + 3 \]

\[ D(\cot(3x+6y), x) \]
\[ = 2 \cdot -3\cot(6y + 3x) - 3 \]

\[ D(\sec(3x+6y), x) \]
\[ = 3\sec(6y + 3x)\tan(6y + 3x) \]

\[ D(\csc(3x+6y), x) \]
\[ = -3\cot(6y + 3x)\csc(6y + 3x) \]

\[ D(\arcsin(3x+6y), x) \]
\[ = \frac{3}{\sqrt{2}^2 - 36y - 36x^2 - 9x + 1} \]

\[ D(\arccos(3x+6y), x) \]
\[ = -\frac{3}{\sqrt{2}^2 - 36y - 36x^2 - 9x + 1} \]

\[ D(\arctan(3x+6y), x) \]
\[ = \frac{3}{2\sqrt{36y + 36x^2 + 9x + 1}} \]

\[ D(\arccot(3x+6y), x) \]
\[ = -\frac{3}{2\sqrt{36y + 36x^2 + 9x + 1}} \]
\[ D(\text{asec}(3x+6y), x) \]
\[
\frac{1}{1 - \frac{(2y + x)^2}{36y^2 + 36x^2y + 9x^2 - 1}}
\]
\[ D(\text{acsc}(3x+6y), x) \]
\[
\frac{-1}{1 - \frac{(2y + x)^2}{36y^2 + 36x^2y + 9x^2 - 1}}
\]
\[ D(\sinh(3x+6y), x) \]
\[
3 \cosh(6y + 3x)
\]
\[ D(\cosh(3x+6y), x) \]
\[
3 \sinh(6y + 3x)
\]
\[ D(\tanh(3x+6y), x) \]
\[
\frac{2}{1 - 3 \tanh(6y + 3x)^2 + 3}
\]
\[ D(\coth(3x+6y), x) \]
\[
\frac{2}{1 - 3 \coth(6y + 3x)^2 + 3}
\]
\[ D(\text{sech}(3x+6y), x) \]
\[
-3 \text{sech}(6y + 3x) \tanh(6y + 3x)
\]
\[ D(\text{csch}(3x+6y), x) \]
\[
-3 \coth(6y + 3x) \csch(6y + 3x)
\]
\[ D(\text{asinh}(3x+6y), x) \]
\[
\frac{3}{1 - \frac{(36y + 36x^2y + 9x^2 + 1)^2}{36y^2 + 36x^2y + 9x^2 + 1}}
\]
\[ D(\text{acosh}(3x+6y), x) \]
\[
\frac{3}{1 - \frac{(36y + 36x^2y + 9x^2 - 1)^2}{36y^2 + 36x^2y + 9x^2 - 1}}
\]
D(\text{atanh}(3x+6y),x)\bigg|_{3} - \frac{\text{------------------------}}{2 \ 2} \frac{36y + 36x y + 9x - 1}{2 \ 2} \frac{36y + 36x y + 9x - 1}{2 \ 2} \\
D(\text{acoth}(3x+6y),x)\bigg|_{3} - \frac{\text{------------------------}}{2 \ 2} \frac{36y + 36x y + 9x - 1}{2 \ 2} \frac{36y + 36x y + 9x - 1}{2 \ 2} \\
D(\text{asech}(3x+6y),x)\bigg|_{1} - \frac{\text{---------------------------------}}{2 \ 2} \frac{(2y + x) / (2y + x)}{2 \ 2} \frac{(2y + x) / (2y + x)}{2 \ 2} \\
D(\text{acsch}(3x+6y),x)\bigg|_{1} - \frac{\text{---------------------------------}}{2 \ 2} \frac{(2y + x) / (2y + x)}{2 \ 2} \frac{(2y + x) / (2y + x)}{2 \ 2} \\

See Also:
\o )\text{show ElementaryFunction}
acos  acosh  acot  acoth  acsc
acsch acsch  acsch  acsc  acsc
atan  atanh  belong?  cos  cosh
cot  coth  csc  csch  exp
iiacos iiacosh iiacot iiacoth iiacsc
iiacsch iiasec iiasech iiasin iiasinh
iitanh iiatanh iiacos iiacosh iiacot
iicoth iiicsc iicsch iicexp iilog
iisec iiserech iisin iisinh iisqrt2
iisqrt3 iiitan iiitanh localReal? log
operator pi  sec  sech  sin
sinh  specialTrigs tan  tanh

— package EF ElementaryFunction —

)abbrev package EF ElementaryFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 April 1995
++ Description:
++ Provides elementary functions over an integral domain.

ElementaryFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: Join(FunctionSpace R, RadicalCategory)

B ==> Boolean
L ==> List
Z ==> Integer
OP ==> BasicOperator
K ==> Kernel F
INV ==> error "Invalid argument"

Exports ==
  with
    exp : F -> F
    ++ exp(x) applies the exponential operator to x
    log : F -> F
    ++ log(x) applies the logarithm operator to x
    sin : F -> F
    ++ sin(x) applies the sine operator to x
    cos : F -> F
    ++ cos(x) applies the cosine operator to x
    tan : F -> F
    ++ tan(x) applies the tangent operator to x
    cot : F -> F
    ++ cot(x) applies the cotangent operator to x
    sec : F -> F
    ++ sec(x) applies the secant operator to x
    csc : F -> F


++ csc(x) applies the cosecant operator to x
asin : F -> F
  ++ asin(x) applies the inverse sine operator to x
acos : F -> F
  ++ acos(x) applies the inverse cosine operator to x
atan : F -> F
  ++ atan(x) applies the inverse tangent operator to x
acot : F -> F
  ++ acot(x) applies the inverse cotangent operator to x
asec : F -> F
  ++ asec(x) applies the inverse secant operator to x
acsc : F -> F
  ++ acsc(x) applies the inverse cosecant operator to x
sinh : F -> F
  ++ sinh(x) applies the hyperbolic sine operator to x
cosh : F -> F
  ++ cosh(x) applies the hyperbolic cosine operator to x
tanh : F -> F
  ++ tanh(x) applies the hyperbolic tangent operator to x
coth : F -> F
  ++ coth(x) applies the hyperbolic cotangent operator to x
sech : F -> F
  ++ sech(x) applies the hyperbolic secant operator to x
csch : F -> F
  ++ csch(x) applies the hyperbolic cosecant operator to x
asinh : F -> F
  ++ asinh(x) applies the inverse hyperbolic sine operator to x
acosh : F -> F
  ++ acosh(x) applies the inverse hyperbolic cosine operator to x
atanh : F -> F
  ++ atanh(x) applies the inverse hyperbolic tangent operator to x
coth : F -> F
  ++ coth(x) applies the inverse hyperbolic cotangent operator to x
asech : F -> F
  ++ asech(x) applies the inverse hyperbolic secant operator to x
acsch : F -> F
  ++ acsch(x) applies the inverse hyperbolic cosecant operator to x
pi : () -> F
  ++ pi() returns the pi operator
belong? : OP -> Boolean
  ++ belong?(p) returns true if operator p is elementary
operator : OP -> OP
  ++ operator(p) returns an elementary operator with the same symbol as p
-- the following should be local, but are conditional
iisqrt2 : () -> F
  ++ iisqrt2() should be local but conditional
iisqrt3 : () -> F
  ++ iisqrt3() should be local but conditional
iiexp : F -> F
  ++ iiexp(x) should be local but conditional
\begin{verbatim}
iilog : F -> F
  ++ iilog(x) should be local but conditional
iisin  : F -> F
  ++ iisin(x) should be local but conditional
iicos  : F -> F
  ++ iicos(x) should be local but conditional
iitanc : F -> F
  ++ iitanc(x) should be local but conditional
iicot  : F -> F
  ++ iicot(x) should be local but conditional
iiisc  : F -> F
  ++ iiisc(x) should be local but conditional
iiasin : F -> F
  ++ iiasin(x) should be local but conditional
iiacos : F -> F
  ++ iiacos(x) should be local but conditional
iiatanc: F -> F
  ++ iiatanc(x) should be local but conditional
iiacot : F -> F
  ++ iiacot(x) should be local but conditional
iiasec : F -> F
  ++ iiasec(x) should be local but conditional
iiacsc : F -> F
  ++ iiacsc(x) should be local but conditional
iisinh : F -> F
  ++ iisinh(x) should be local but conditional
iicosh : F -> F
  ++ iicosh(x) should be local but conditional
iiitanh: F -> F
  ++ iiitanh(x) should be local but conditional
iiicoth : F -> F
  ++ iiicoth(x) should be local but conditional
iisech : F -> F
  ++ iisech(x) should be local but conditional
iicsch : F -> F
  ++ iicsch(x) should be local but conditional
iiasinh : F -> F
  ++ iiasinh(x) should be local but conditional
iiacosh : F -> F
  ++ iiacosh(x) should be local but conditional
iiatanh: F -> F
  ++ iiatanh(x) should be local but conditional
iiacoth : F -> F
  ++ iiacoth(x) should be local but conditional
iiasech : F -> F
  ++ iiasech(x) should be local but conditional
iiacsch : F -> F
  ++ iiacsch(x) should be local but conditional
\end{verbatim}
specialTrigs:(F, L Record(func:F,pole:B)) -> Union(F, "failed")
++ specialTrigs(x,l) should be local but conditional
localReal?: F -> Boolean
++ localReal?(x) should be local but conditional

Implementation ==> add
ipi : List F -> F
iexp : F -> F
ilog : F -> F
iiilog : F -> F
isin : F -> F
icos : F -> F
itan : F -> F
icot : F -> F
isec : F -> F
icsc : F -> F
iasin : F -> F
iacos : F -> F
iatan : F -> F
iacot : F -> F
iasec : F -> F
iacsc : F -> F
isinh : F -> F
icosh : F -> F
itanh : F -> F
icotanh : F -> F
iasech : F -> F
iacsch : F -> F
iisqrt1 : () -> F
valueOrPole : Record(func:F, pole:B) -> F

oppi := operator("pi": Symbol)$CommonOperators
oplog := operator("log": Symbol)$CommonOperators
opexp := operator("exp": Symbol)$CommonOperators
opsin := operator("sin": Symbol)$CommonOperators
opcos := operator("cos": Symbol)$CommonOperators
optan := operator("tan": Symbol)$CommonOperators
opcot := operator("cot": Symbol)$CommonOperators
opsec := operator("sec": Symbol)$CommonOperators
opcsc := operator("csc": Symbol)$CommonOperators
opasin := operator("asin": Symbol)$CommonOperators
opacos := operator("acos"::Symbol)$CommonOperators
opatan := operator("atan"::Symbol)$CommonOperators
opacot := operator("acot"::Symbol)$CommonOperators
opasec := operator("asec"::Symbol)$CommonOperators
opacsc := operator("acsc"::Symbol)$CommonOperators
opsinh := operator("sinh"::Symbol)$CommonOperators
opcosh := operator("cosh"::Symbol)$CommonOperators
optanh := operator("tanh"::Symbol)$CommonOperators
opcoth := operator("coth"::Symbol)$CommonOperators
opsech := operator("sech"::Symbol)$CommonOperators
opcsch := operator("csch"::Symbol)$CommonOperators
opasinh := operator("asinh"::Symbol)$CommonOperators
opacosh := operator("acosh"::Symbol)$CommonOperators
opatanh := operator("atanh"::Symbol)$CommonOperators
opacoth := operator("acoth"::Symbol)$CommonOperators
opasech := operator("asech"::Symbol)$CommonOperators
opacsch := operator("acsch"::Symbol)$CommonOperators

-- Pi is a domain...
Pie, isqrt1, isqrt2, isqrt3: F

-- following code is conditionalized on arbitraryPrecision to recompute in
-- case user changes the precision

if R has TranscendentalFunctionCategory then
    Pie := pi()$R :: F
else
    Pie := kernel(oppi, nil()$List(F))

if R has TranscendentalFunctionCategory and R has arbitraryPrecision then
    pi() == pi()$R :: F
else
    pi() == Pie

if R has imaginary: () -> R then
    isqrt1 := imaginary()$R :: F
else
    isqrt1 := sqrt(-1::F)

if R has RadicalCategory then
    isqrt2 := sqrt(2::R)::F
    isqrt3 := sqrt(3::R)::F
else
    isqrt2 := sqrt(2::F)
    isqrt3 := sqrt(3::F)

iisqrt1() == isqrt1

if R has RadicalCategory and R has arbitraryPrecision then
    iisqrt2() == sqrt(2::R)::F
    iisqrt3() == sqrt(3::R)::F
else
iisqrt2() == isqrt2
iisqrt3() == isqrt3

ipi l == pi()
log x == oplog x
exp x == opexp x
sin x == opsin x
cos x == opcos x
tan x == optan x
cot x == opcot x
sec x == opsec x
csc x == opcsc x
asin x == opasin x
acos x == opacos x
atan x == opatan x
acot x == opacot x
asec x == opasec x
acsc x == opacsc x
sinh x == opsinh x
cosh x == opcosh x		tanh x == optanh x
coth x == opcot x
sech x == opsech x
csch x == opcsch x
asinh x == opasinh x
acosh x == opacosh x
atanh x == opatanh x
acoth x == opacoth x
asech x == opasech x
acsch x == opacsch x

kernel x == retract(x)@K

posrem(n, m) == ((r := n rem m) < 0 => r + m; r)
valueOrPole rec == (rec.pole => INV; rec.func)
belong? op == has?(op, "elem")

operator op ==
is?(op, "pi"::Symbol) => oppi
is?(op, "log"::Symbol) => oplog
is?(op, "exp"::Symbol) => opexp
is?(op, "sin"::Symbol) => opsin
is?(op, "cos"::Symbol) => opcos
is?(op, "tan"::Symbol) => optan
is?(op, "cot"::Symbol) => opcot
is?(op, "sec"::Symbol) => opsec
is?(op, "csc"::Symbol) => opcsc
is?(op, "asin"::Symbol) => opasin
is?(op, "acos"::Symbol) => opacos
is?(op, "atan"::Symbol) => opatan
is?(op, "acot"::Symbol) => opacot
is?(op, "asec": Symbol) => opasec
is?(op, "acsc": Symbol) => opacsc
is?(op, "sinh": Symbol) => opsinh
is?(op, "cosh": Symbol) => opcosh
is?(op, "tanh": Symbol) => optanh
is?(op, "coth": Symbol) => opcoth
is?(op, "sech": Symbol) => opsech
is?(op, "csch": Symbol) => opcsch
is?(op, "asinh": Symbol) => opasinh
is?(op, "acosh": Symbol) => opacosh
is?(op, "atanh": Symbol) => opatanh
is?(op, "acoth": Symbol) => opacoth
is?(op, "asech": Symbol) => opasech
is?(op, "acsch": Symbol) => opacsch
error "Not an elementary operator"

dropfun x ==
  (k := retractIfCan(x)@Union(K, "failed")) case "failed" or
  empty?(argument(k::K)) => 0
  first argument(k::K)

if R has RetractableTo Z then
  specialTrigs(x, values) ==
    (r := retractIfCan(y := x/pi())@Union(Fraction Z, "failed"))
    case "failed" => "failed"
    q := r::Fraction(Integer)
    m := minIndex values
    (n := retractIfCan(q)@Union(Z, "failed")) case Z =>
      even?(n::Z) => valueOrPole(values.m)
      valueOrPole(values.(m+1))
    (n := retractIfCan(2*q)@Union(Z, "failed")) case Z =>
      one?(s := posrem(n::Z, 4)) => valueOrPole(values.(m+2))
      valueOrPole(values.(m+3))
    (n := retractIfCan(3*q)@Union(Z, "failed")) case Z =>
      one?(s := posrem(n::Z, 6)) => valueOrPole(values.(m+4))
      valueOrPole(values.(m+5))
    (n := retractIfCan(4*q)@Union(Z, "failed")) case Z =>
      one?(s := posrem(n::Z, 8)) => valueOrPole(values.(m+8))
      valueOrPole(values.(m+9))
    (n := retractIfCan(6*q)@Union(Z, "failed")) case Z =>
      one?(s := posrem(n::Z, 12)) => valueOrPole(values.(m+12))

--
s = 7 => valueOrPole(values.(m+14))

valueOrPole(values.(m+15))

"failed"

else specialTrigs(x, values) == "failed"

isin x ==
    zero? x => 0
    y := dropfun x
    is?(x, opasin) => y
    is?(x, opacos) => sqrt(1 - y**2)
    is?(x, opatan) => y / sqrt(1 + y**2)
    is?(x, opacot) => inv sqrt(1 + y**2)
    is?(x, opasec) => sqrt(y**2 - 1) / y
    is?(x, opacsc) => inv y
    h := inv(2::F)
    s2 := h * iisqrt2()
    s3 := h * iisqrt3()
    u := specialTrigs(x, [[0,false], [0,false], [1,false], [-1,false],
                        [s3,false], [s3,false], [-s3,false], [-s3,false],
                        [s2,false], [s2,false], [-s2,false], [-s2,false],
                        [h,false], [h,false], [-h,false], [-h,false]])
    u case F => u :: F
    kernel(opsin, x)

icos x ==
    zero? x => 1
    y := dropfun x
    is?(x, opasin) => sqrt(1 - y**2)
    is?(x, opacos) => y
    is?(x, opatan) => inv sqrt(1 + y**2)
    is?(x, opacot) => y / sqrt(1 + y**2)
    is?(x, opasec) => y
    h := inv(2::F)
    s2 := h * iisqrt2()
    s3 := h * iisqrt3()
    u := specialTrigs(x, [[1,false],[-1,false], [0,false], [0,false],
                          [h,false],[-h,false],[h,false],[-h,false],[h,false],
                          [s2,false],[-s2,false],[-s2,false],[s2,false],
                          [s3,false],[-s3,false],[-s3,false],[s3,false]])
    u case F => u :: F
    kernel(opcos, x)

itan x ==
    zero? x => 0
    y := dropfun x
    is?(x, opasin) => y / sqrt(1 - y**2)
    is?(x, opacos) => sqrt(1 - y**2) / y
    is?(x, opatan) => y
CHAPTER 6. CHAPTER E

is?(x, opacot) => inv y
is?(x, opasec) => sqrt(y**2 - 1)
is?(x, opacsc) => inv sqrt(y**2 - 1)
s33 := (s3 := iisqrt3()) / (3::F)
u := specialTrigs(x, [[0,false], [0,false], [0,true], [0,true],
[s3,false], [-s3,false], [s3,false], [-s3,false],
[1,false], [-1,false], [1,false], [-1,false],
[s3,false], [-s3,false], [s3,false], [-s3,false])
u case F => u :: F
kernel(optan, x)

icot x ==
zero? x => INV
y := dropfun x
is?(x, opasin) => sqrt(1 - y**2) / y
is?(x, opacos) => y / sqrt(1 - y**2)
is?(x, opatan) => inv y
is?(x, opacot) => y
is?(x, opasec) => inv sqrt(y**2 - 1)
is?(x, opacsc) => sqrt(y**2 - 1)
s33 := (s3 := iisqrt3()) / (3::F)
u := specialTrigs(x, [[0,true], [0,true], [0,false], [0,false],
[s3,false], [-s3,false], [s3,false], [-s3,false],
[1,false], [-1,false], [1,false], [-1,false],
[s3,false], [-s3,false], [s3,false], [-s3,false]])
u case F => u :: F
kernel(opcot, x)

isec x ==
zero? x => 1
y := dropfun x
is?(x, opasin) => inv sqrt(1 - y**2)
is?(x, opacos) => inv y
is?(x, opatan) => sqrt(1 + y**2)
is?(x, opacot) => sqrt(1 + y**2) / y
is?(x, opasec) => y
is?(x, opacsc) => y / sqrt(y**2 - 1)
s2 := iisqrt2()
s3 := 2 * iisqrt3() / (3::F)
h := 2::F
u := specialTrigs(x, [[1,false],[-1,false],[0,true],[0,true],
[h,false], [-h,false], [-h,false], [h,false],
s2,false], [-s2,false], [-s2,false], [s2,false],
[s3,false], [-s3,false], [s3,false], [-s3,false]])
u case F => u :: F
kernel(opsec, x)

icsc x ==
zero? x => INV
y := dropfun x
is?(x, opasin) => inv y
is?(x, opacos) => inv sqrt(1 - y**2)
is?(x, opatan) => sqrt(1 + y**2) / y
is?(x, opacot) => sqrt(1 + y**2)
is?(x, opasec) => y / sqrt(y**2 - 1)
is?(x, opacsc) => y
s2 := iisqrt2()
s3 := 2 * iisqrt3() / (3::F)
h := 2::F
u := specialTrigs(x, [[0,true], [0,true], [1,false], [-1,false],
    [s3,false], [s3,false], [-s3,false], [-s3,false],
    [s2,false], [s2,false], [-s2,false], [-s2,false],
    [h,false], [h,false], [-h,false], [-h,false]])

{u case F => u :: F
kernel(opcsc, x)}

iasin x ==
zero? x => 0
-- one? x => pi() / (2::F)
(x = 1) => pi() / (2::F)
x = -1 => - pi() / (2::F)
y := dropfun x
is?(x, opsin) => y
is?(x, opcos) => pi() / (2::F) - y
kernel(opasin, x)

iacos x ==
zero? x => pi() / (2::F)
-- one? x => 0
(x = 1) => 0
x = -1 => pi()
y := dropfun x
is?(x, opsin) => pi() / (2::F) - y
is?(x, opcos) => y
kernel(opacos, x)

iatan x ==
zero? x => 0
-- one? x => pi() / (4::F)
(x = 1) => pi() / (4::F)
x = -1 => - pi() / (4::F)
x = (r3:=iisqrt3()) => pi() / (3::F)
-- one?(x*r3) => pi() / (6::F)
(x*r3) = 1 => pi() / (6::F)
y := dropfun x
is?(x, optan) => y
is?(x, opcot) => pi() / (2::F) - y
kernel(opatan, x)

iacot x ==
iasec x ==
zero? x => INV
-- one? x => 0
(x = 1) => pi()
(x = -1) => -pi()
y := dropfun x
is?(x, opsec) => y
is?(x, opcsc) => pi() / (2::F) - y
kernel(opasec, x)

iacsc x ==
zero? x => INV
-- one? x => pi() / (2::F)
(x = 1) => pi() / (2::F)
(x = -1) => -pi() / (2::F)
y := dropfun x
is?(x, opsec) => pi() / (2::F) - y
is?(x, opcsc) => y
kernel(opacsc, x)

isinh x ==
zero? x => 0
y := dropfun x
is?(x, opasinh) => y
is?(x, opacosh) => sqrt(y**2 - 1)
is?(x, opatanh) => y / sqrt(1 - y**2)
is?(x, opcoth) => -inv sqrt(y**2 - 1)
is?(x, opasech) => sqrt(1 - y**2) / y
is?(x, opacsch) => inv y
kernel(opsinh, x)

icosh x ==
zero? x => 1
y := dropfun x
is?(x, opasinh) => sqrt(y**2 + 1)
is?(x, opacosh) => y
is?(x, opatanh) => inv sqrt(1 - y**2)

is?(x, opacoth) => y / sqrt(y**2 - 1)

is?(x, opasech) => inv y

is?(x, opacsch) => sqrt(y**2 + 1) / y

kernel(opcosh, x)

itanh x ==
zero? x => 0

y := dropfun x

is?(x, opasinh) => y / sqrt(y**2 + 1)

is?(x, opacosh) => sqrt(y**2 - 1) / y

is?(x, opatanh) => y

is?(x, opacoth) => inv y

is?(x, opasech) => sqrt(1 - y**2)

is?(x, opacsch) => inv sqrt(y**2 + 1)

kernel(optanh, x)

icoth x ==
zero? x => INV

y := dropfun x

is?(x, opasinh) => sqrt(y**2 + 1) / y

is?(x, opacosh) => y / sqrt(y**2 - 1)

is?(x, opatanh) => inv y

is?(x, opacoth) => y

is?(x, opasech) => inv sqrt(1 - y**2)

is?(x, opacsch) => sqrt(y**2 + 1)

kernel(opcoth, x)

isech x ==
zero? x => 1

y := dropfun x

is?(x, opasinh) => inv sqrt(y**2 + 1)

is?(x, opacosh) => inv y

is?(x, opatanh) => sqrt(1 - y**2)

is?(x, opacoth) => sqrt(y**2 - 1) / y

is?(x, opasech) => y

is?(x, opacsch) => y / sqrt(1 - y**2)

is?(x, opacsch) => y

kernel(opsech, x)

icsch x ==
zero? x => INV

y := dropfun x

is?(x, opasinh) => inv y

is?(x, opacosh) => inv sqrt(y**2 - 1)

is?(x, opatanh) => sqrt(y**2 - 1) / y

is?(x, opacoth) => - sqrt(y**2 - 1)

is?(x, opasech) => y / sqrt(1 - y**2)

is?(x, opacsch) => y

kernel(opcsch, x)
\(\text{iasinh} \ x \ = \ \text{is?}(x, \ \text{opsinh}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opsinh}, x)\)

\(\text{iacosh} \ x \ = \ \text{is?}(x, \ \text{opcosh}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opcosh}, x)\)

\(\text{iatanh} \ x \ = \ \text{is?}(x, \ \text{optanh}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opatanh}, x)\)

\(\text{iacoth} \ x \ = \ \text{is?}(x, \ \text{opcoth}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opacoth}, x)\)

\(\text{iasech} \ x \ = \ \text{is?}(x, \ \text{opsech}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opasech}, x)\)

\(\text{iacsch} \ x \ = \ \text{is?}(x, \ \text{opcsch}) \Rightarrow \text{first argument kernel } x\)
\(\text{kernel}(\text{opacsch}, x)\)

\(\text{iexp} \ x \ =
\text{zero?} \ x \ => \ 1\)
\(\text{is?}(x, \ \text{oplog}) \Rightarrow \text{first argument kernel } x\)
\(\text{x} < 0 \ \text{and empty? variables } x \Rightarrow \text{inv iexp}(-x)\)
\(h := \text{inv}(2::F)\)
\(i := \text{iisqrt}1()\)
\(s2 := h * \text{iisqrt}2()\)
\(s3 := h * \text{iisqrt}3()\)
\(u := \text{specialTrigs}(x / i, [[1, \text{false}], [-1, \text{false}], [i, \text{false}], [-i, \text{false}],
[h + i * s3, \text{false}], [h - i * s3, \text{false}],
[s2 + i * s2, \text{false}], [-s2 + i * s2, \text{false}],
[-s2 - i * s2, \text{false}], [s2 - i * s2, \text{false}],
[s3 + i * h, \text{false}], [-s3 + i * h, \text{false}],
[-s3 - i * h, \text{false}], [s3 - i * h, \text{false}]])\)
\(u \ \text{case } F \Rightarrow u :: F\)
\(\text{kernel}(\text{opexp}, x)\)

-- THIS DETERMINES WHEN TO PERFORM THE \(\log \ \text{exp} \ f -> f\) SIMPLIFICATION
-- CURRENT BEHAVIOR:
-- IF R IS COMPLEX(S) THEN ONLY ELEMENTS WHICH ARE RETRACTABLE TO R
-- AND EQUAL TO THEIR CONJUGATES ARE DEEMED REAL (OVERRESTRICTIVE FOR NOW)
-- OTHERWISE (e.g. R = INT OR FRAC INT), ALL THE ELEMENTS ARE DEEMED REAL

if (R \text{has imaginary:()} \Rightarrow R) \text{ and } (R \text{has conjugate: } R \Rightarrow R) \text{ then}
\(\text{localReal? } x \ =\)
\((u := \text{retractIfCan}(x)@\text{Union}(R, "failed")) \text{ case } R\)
and (u::R) = \text{conjugate}(u::R)
else localReal? x == true

iiilog x ==
    zero? x => INV
    -- one? x => 0
    (x = 1) => 0
    (u := isExpt(x, opexp)) case Record(var:K, exponent:Integer) =>
        rec := u::Record(var:K, exponent:Integer)
        arg := first argument(rec.var);
        localReal? arg => rec.exponent * first argument(rec.var);
        iilog x
    iilog x

iiilog x ==
    -- ((num1 := one? (num := numer x)) or num = -1) and (den := denom x) ^= 1
    ((num1 := ((num := numer x) = 1)) or num = -1) and (den := denom x) ^= 1
    and empty? variables x => - kernel(oplog, (num1 => den; -den)::F)
    kernel(oplog, x)

if R has ElementaryFunctionCategory then
    iilog x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iiilog x
        log(r::R)::F

iiexp x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iexp x
    exp(r::R)::F

else
    iilog x == iiilog x
    iiexp x == iexp x

if R has TrigonometricFunctionCategory then
    iisin x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isin x
        sin(r::R)::F

iicos x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icos x
    cos(r::R)::F

ittan x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => itan x
    tan(r::R)::F

iicot x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icot x
    cot(r::R)::F
\[
\begin{align*}
iisec\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isec x sec(r::R)::F} \\
iiscsc\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icsc x csc(r::R)::F} \\
\text{else} \\
iisin\ x\ &=\ isin\ x \\
iicos\ x\ &=\ icos\ x \\
iitan\ x\ &=\ itan\ x \\
iicot\ x\ &=\ icot\ x \\
iisec\ x\ &=\ isec\ x \\
iiscsc\ x\ &=\ icsc\ x \\
\text{if R has ArcTrigonometricFunctionCategory then} \\
iiasin\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasin x asin(r::R)::F} \\
iiacos\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacos x acos(r::R)::F} \\
iiatan\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iatan x atan(r::R)::F} \\
iiacot\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacot x acot(r::R)::F} \\
iiasec\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasec x asec(r::R)::F} \\
iiacsc\ x\ &=\ \text{(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsc x acsc(r::R)::F} \\
\text{else} \\
iiasin\ x\ &=\ iasin\ x \\
iiacos\ x\ &=\ iacos\ x \\
iiatan\ x\ &=\ iatan\ x \\
iiacot\ x\ &=\ iacot\ x \\
iiasec\ x\ &=\ iasec\ x \\
iiacsc\ x\ &=\ iacsc\ x \\
\text{if R has HyperbolicFunctionCategory then}
\end{align*}
\]
PACKAGE EF ELEMENTARYFUNCTION

\begin{verbatim}
ii\text{isinh} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{isinh} x
  \sinh(r::R)::F

ii\text{icosh} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{icosh} x
  \cosh(r::R)::F

ii\text{itanh} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{itanh} x
  \tanh(r::R)::F

ii\text{icoth} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{icoth} x
  \coth(r::R)::F

ii\text{isech} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{isech} x
  \sech(r::R)::F

ii\text{icsch} x ==
  (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{icsch} x
  \csch(r::R)::F

else
  ii\text{isinh} x == \text{isinh} x
  ii\text{icosh} x == \text{icosh} x
  ii\text{itanh} x == \text{itanh} x
  ii\text{icoth} x == \text{icoth} x
  ii\text{isech} x == \text{isech} x
  ii\text{icsch} x == \text{icsch} x

if R has ArcHyperbolicFunctionCategory then
  ii\text{iasinh} x ==
    (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{iasinh} x
    \text{asinh}(r::R)::F

  ii\text{iacosh} x ==
    (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{iacosh} x
    \text{acosh}(r::R)::F

  ii\text{atanh} x ==
    (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{atanh} x
    \text{atanh}(r::R)::F

  ii\text{acoth} x ==
    (r:=\text{retractIfCan}(x)@\text{Union}(R, "failed")) case "failed" => \text{acoth} x
    \text{acoth}(r::R)::F

  ii\text{asech} x ==
\end{verbatim}
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasech x
asech(r::R)::F

iiacsch x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsch x
acsch(r::R)::F

else
iiasinh x == iasinh x
iiacosh x == iacosh x
iiatanh x == iatanh x
iiacoth x == iacoth x
iiasech x == iasech x
iiacsch x == iacsch x

import BasicOperatorFunctions1(F)
evaluate(oppi, ipi)
evaluate(oplog, iilog)
evaluate(opexp, iiexp)
evaluate(opsin, iisin)
evaluate(opcos, iicos)
evaluate(optan, iitan)
evaluate(opcot, iicot)
evaluate(opsec, iisec)
evaluate(opcsc, iicsc)
evaluate(opasin, iiasin)
evaluate(opacos, iiacos)
evaluate(opatan, iiatan)
evaluate(opacot, iiacot)
evaluate(opasech, iiasech)
evaluate(opacsch, iiacsch)
evaluate(opasinh, iiasinh)
evaluate(opacosh, iiacosh)
evaluate(opatanh, iiatanh)
evaluate(opacoth, iiacoth)
evaluate(opasech, iiasech)
evaluate(opacsch, iiacsch)
derivative(opexp, exp)
derivative(oplog, inv)
derivative(opsin, cos)
derivative(opcos,(x:F):F +-> - sin x)
derivative(optan,(x:F):F +-> 1 + tan(x)**2)
derivative(opcot,(x:F):F +-> - 1 - cot(x)**2)
derivative(opsec,(x:F):F +-> tan(x) * sec(x))
derivative(opcsc,(x:F):F +-> - cot(x) * csc(x))
derivative(opasin,(x:F):F +-> inv sqrt(1 - x**2))
derivative(opacos,(x:F):F +-> - inv sqrt(1 - x**2))
derivative(opatan,(x:F):F +-> inv(1 + x**2))
derivative(opacot,(x:F):F +-> - inv(1 + x**2))
derivative(opasec,(x:F):F +-> inv(x * sqrt(x**2 - 1)))
derivative(opacsc,(x:F):F +-> - inv(x * sqrt(x**2 - 1)))
derivative(opsinh, cosh)
derivative(opcosh, sinh)
derivative(optanh,(x:F):F +-> 1 - tanh(x)**2)
derivative(opcoth,(x:F):F +-> 1 - coth(x)**2)
derivative(opsech,(x:F):F +-> - tanh(x) * sech(x))
derivative(opcsch,(x:F):F +-> - coth(x) * csch(x))
derivative(opasinh,(x:F):F +-> inv sqrt(1 + x**2))
derivative(opacosh,(x:F):F +-> inv sqrt(x**2 - 1))
derivative(opatanh,(x:F):F +-> inv(1 - x**2))
derivative(opacoth,(x:F):F +-> inv(1 - x**2))
derivative(opasech,(x:F):F +-> - inv(x * sqrt(1 - x**2)))
derivative(opacsch,(x:F):F +-> - inv(x * sqrt(1 + x**2)))

— EF.dotabb —

"EF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EF" -> "FS"

— package DEFINTEF ElementaryFunctionDefiniteIntegration —

)set break resume
)sys rm -f ElementaryFunctionDefiniteIntegration.output
)spool ElementaryFunctionDefiniteIntegration.output
)set message test on
)set message auto off
)clear all
ElementaryFunctionDefiniteIntegration provides functions to compute definite integrals of elementary functions.

See Also:
o )show ElementaryFunctionDefiniteIntegration

---

**Exports:**
innerint integrate integrate

---

+ Author: Manuel Bronstein
ElementaryFunctionDefiniteIntegration(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
           AlgebraicallyClosedFunctionSpace R)

  B ==> Boolean
  SE ==> Symbol
  Z ==> Integer
  P ==> SparseMultivariatePolynomial(R, K)
  K ==> Kernel F
  UP ==> SparseUnivariatePolynomial F
  OFE ==> OrderedCompletion F
  U ==> Union(f1:OFE, f2:List OFE, fail:"failed", pole:"potentialPole")

Exports ==> with
  integrate: (F, SegmentBinding OFE) -> U
  ++ integrate(f, x = a..b) returns the integral of
  ++ \spad{f(x)dx} from a to b.
  ++ Error: if f has a pole for x between a and b.
  integrate: (F, SegmentBinding OFE, String) -> U
  ++ integrate(f, x = a..b, "noPole") returns the
  ++ integral of \spad{f(x)dx} from a to b.
  ++ If it is not possible to check whether f has a pole for x
  ++ between a and b (because of parameters), then this function
  ++ will assume that f has no such pole.
  ++ Error: if f has a pole for x between a and b or
  ++ if the last argument is not "noPole".
  innerint: (F, SE, OFE, OFE, B) -> U
  ++ innerint(f, x, a, b, ignore?) should be local but conditional

Implementation ==> add
  import ElementaryFunctionSign(R, F)
  import DefiniteIntegrationTools(R, F)
  import FunctionSpaceIntegration(R, F)

  polyIfCan : (P, K) -> Union(UP, "failed")
  int : (F, SE, OFE, OFE, B) -> U
  nopole : (F, SE, K, OFE, OFE) -> U
  checkFor0 : (P, K, OFE, OFE) -> Union(B, "failed")
  checkSMP : (P, SE, K, OFE, OFE) -> Union(B, "failed")
  checkForPole: (F, SE, K, OFE, OFE) -> Union(B, "failed")
  posit : (F, SE, K, OFE, OFE) -> Union(B, "failed")
negat : (F, SE, K, OFE, OFE) -> Union(B, "failed")
moreThan : (OFE, Fraction Z) -> Union(B, "failed")

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
and F has SpecialFunctionCategory then
import PatternMatchIntegration(R, F)

innerint(f, x, a, b, ignor?) ==
((u := int(f, x, a, b, ignor?)) case f1) or (u case f2)
or ((v := pmintegrate(f, x, a, b)) case "failed") => u
[v::F::OFE]
else
innerint(f, x, a, b, ignor?) == int(f, x, a, b, ignor?)

integrate(f:F, s:SegmentBinding OFE) ==
innerint(f, variable s, lo segment s, hi segment s, false)

integrate(f:F, s:SegmentBinding OFE, str:String) ==
innerint(f, variable s, lo segment s, hi segment s, ignore? str)

int(f, x, a, b, ignor?) ==
a = b => [0::OFE]

k := kernel(x)@Kernel(F)
(z := checkForPole(f, x, k, a, b)) case "failed" =>
ignor? => nopole(f, x, k, a, b)
["potentialPole"]
z::B => error "integrate: pole in path of integration"

nopole(f, x, k, a, b)

checkForPole(f, x, k, a, b) ==
((u := checkFor0(d := denom f, k, a, b)) case "failed") or (u::B) => u
((u := checkSMP(d, x, k, a, b)) case "failed") or (u::B) => u
checkSMP(numer f, x, k, a, b)

-- true if p has a zero between a and b exclusive
checkFor0(p, x, a, b) ==
(u := polyIfCan(p, x)) case UP => checkForZero(u::UP, a, b, false)
(v := isTimes p) case List(P) =>
for t in v::List(P) repeat
((w := checkFor0(t, x, a, b)) case "failed") or (w::B) => return w
false
(r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"
k := r::K

-- functions with no real zeros
is?(k, "exp":SE) or is?(k, "acot":SE) or is?(k, "cosh":SE) => false

-- special case for log
is?(k, "log":SE) =>
(w := moreThan(b, 1)) case "failed" or not(w::B) => w
moreThan(-a, -1)
"failed"

-- returns true if \( a > b \), false if \( a < b \), "failed" if can't decide
moreThan(a, b) ==
  (r := retractIfCan(a)@Union(F, "failed")) case "failed" => -- infinite
    whatInfinity(a) > 0
  (u := retractIfCan(r::F)@Union(Fraction Z, "failed")) case "failed" =>
    "failed"
  u::Fraction(Z) > b

-- true if \( p \) has a pole between \( a \) and \( b \)
checkSMP(p, x, k, a, b) ==
  (u := polyIfCan(p, k)) case UP => false
  (v := isTimes p) case List(P) =>
    for t in v::List(P) repeat
      (w := checkSMP(t, x, k, a, b)) case "failed" or (w::B) => return w
      false
  (v := isPlus p) case List(P) =>
    n := 0 -- number of summand having a pole
    for t in v::List(P) repeat
      (w := checkSMP(t, x, k, a, b)) case "failed" => return w
      if w::B then n := n + 1
    zero? n => false -- no summand has a pole
    one? n => true -- only one summand has a pole
    (n = 1) => true -- only one summand has a pole
    "failed" -- at least 2 summands have a pole
  (r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"
  kk := r::K
  -- nullary operators have no poles
  nullary? operator kk => false
  f := first argument kk
  -- functions which are defined over all the reals:
  is?(kk, "exp"::SE) or is?(kk, "sin"::SE) or is?(kk, "cos"::SE)
  or is?(kk, "sinh"::SE) or is?(kk, "cosh"::SE) or is?(kk, "tanh"::SE)
  or is?(kk, "sech"::SE) or is?(kk, "atan"::SE) or is?(kk, "acot"::SE)
  or is?(kk, "asinh"::SE) => checkForPole(f, x, k, a, b)
  -- functions which are defined on \((-1, +1)\):
  is?(kk, "asin"::SE) or is?(kk, "acos"::SE) or is?(kk, "atanh"::SE) =>
    ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
    ((w := posit(f - 1, x, k, a, b)) case "failed") or (w::B) => w
    negat(f + 1, x, k, a, b)
  -- functions which are defined on \((+1, +\infty)\):
  is?(kk, "acosh"::SE) =>
    ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
    negat(f - 1, x, k, a, b)
  -- functions which are defined on \((-\infty, +\infty)\):
  is?(kk, "log"::SE) =>
    ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
    negat(f, x, k, a, b)
  "failed"
-- returns true if it is certain that f takes at least one strictly positive
-- value for x in (a,b), false if it is certain that f takes no strictly
-- positive value in (a,b), "failed" otherwise
-- f must be known to have no poles in (a,b)
posit(f, x, k, a, b) ==
  z :=
  (r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
  sign(f, x, r::F, "right")
  (b1 := z case Z) and z::Z > 0 => true
  z :=
  (r := retractIfCan(b)@Union(F, "failed")) case "failed" => sign(f, x, b)
  sign(f, x, r::F, "left")
  (b2 := z case Z) and z::Z > 0 => true
  b1 and b2 =>
  ((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
  false
  "failed"

-- returns true if it is certain that f takes at least one strictly negative
-- value for x in (a,b), false if it is certain that f takes no strictly
-- negative value in (a,b), "failed" otherwise
-- f must be known to have no poles in (a,b)
negat(f, x, k, a, b) ==
  z :=
  (r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
  sign(f, x, r::F, "right")
  (b1 := z case Z) and z::Z < 0 => true
  z :=
  (r := retractIfCan(b)@Union(F, "failed")) case "failed" => sign(f, x, b)
  sign(f, x, r::F, "left")
  (b2 := z case Z) and z::Z < 0 => true
  b1 and b2 =>
  ((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
  false
  "failed"

-- returns a UP if p is only a poly w.r.t. the kernel x
polyIfCan(p, x) ==
  q := univariate(p, x)
  ans:UP := 0
  while q ^= 0 repeat
    member?(x, tower(c := leadingCoefficient(q)::F)) => return "failed"
    ans := ans + monomial(c, degree q)
    q := reductum q
  ans

-- integrate f for x between a and b assuming that f has no pole in between
nopole(f, x, k, a, b) ==
  (u := integrate(f, x)) case F =>
(v := computeInt(k, u::F, a, b, false)) case "failed" => ["failed"]
[v::OFE]
ans := empty(List(OFE))
for g in u::List(F) repeat
  (v := computeInt(k, g, a, b, false)) case "failed" => return ["failed"]
  ans := concat_!(ans, [v::OFE])
[ans]

---

— DEFINTEF.dotabb —

"DEFINTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEFINTEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"DEFINTEF" -> "ACFS"

---

package LODEEF ElementaryFunctionLODESolver

— ElementaryFunctionLODESolver.input —

)set break resume
)sys rm -f ElementaryFunctionLODESolver.output
)spool ElementaryFunctionLODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionLODESolver
--E 1

)spool
)lisp (bye)

---

— ElementaryFunctionLODESolver.help —

====================================================================
ElementaryFunctionLODESolver examples
====================================================================
ElementaryFunctionLODESolver provides the top-level functions for finding closed form solutions of linear ordinary differential equations and initial value problems.

See Also:
- )show ElementaryFunctionLODESolver

---

**ElementaryFunctionLODESolver (LODEEF)**

Exports:
solve

— package LODEEF ElementaryFunctionLODESolver —

)abbrev package LODEEF ElementaryFunctionLODESolver
++ Author: Manuel Bronstein
++ Date Created: 3 February 1994
++ Date Last Updated: 9 March 1994
++ Description:
++ \spad{ElementaryFunctionLODESolver} provides the top-level
++ functions for finding closed form solutions of linear ordinary
++ differential equations and initial value problems.

ElementaryFunctionLODESolver(R, F, L): Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
    LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory, PrimitiveFunctionCategory)
L: LinearOrdinaryDifferentialOperatorCategory F
SY ==> Symbol
N ==> NonNegativeInteger
K ==> Kernel F
V ==> Vector F
M ==> Matrix F
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
UPUP==> SparseUnivariatePolynomial RF
P ==> SparseMultivariatePolynomial(R, K)
P2 ==> SparseMultivariatePolynomial(P, K)
LQ ==> LinearOrdinaryDifferentialOperator1 RF
REC ==> Record(particular: F, basis: List F)
U ==> Union(REC, "failed")
ALGOP ==> "%alg"

Exports ==> with
solve: (L, F, SY) -> U
++ solve(op, g, x) returns either a solution of the ordinary differential
++ equation \( \text{op y = g} \) or "failed" if no non-trivial solution can be
++ found; When found, the solution is returned in the form
++ \( \text{spad[h, [b1,...,bm]]} \) where \( \text{spad(h)} \) is a particular solution and
++ and \( \text{spad[bi,...,bm]} \) are linearly independent solutions of the
++ associated homogenous equation \( \text{spad[op y = 0]} \).
++ A full basis for the solutions of the homogenous equation
++ is not always returned, only the solutions which were found;
++ \( \text{spad(x)} \) is the dependent variable.
solve: (L, F, SY, F, List F) -> Union(F, "failed")
++ solve(op, g, x, a, \{y0,...,ym\}) returns either the solution
++ of the initial value problem \( \text{op y = g, y(a) = y0, y'(a) = y1,...} \)
++ or "failed" if the solution cannot be found;
++ \( \text{spad(x)} \) is the dependent variable.

Implementation ==> add
import Kovacic(F, UP)
import ODETools(F, L)
import RationalLODE(F, UP)
import RationalRicDE(F, UP)
import ODEIntegration(R, F)
import ConstantLODE(R, F, L)
import IntegrationTools(R, F)
import ReductionOfOrder(F, L)
import ReductionOfOrder(RF, LQ)
import PureAlgebraicIntegration(R, F, L)
import FunctionSpacePrimitiveElement(R, F)
import LinearSystemMatrixPackage(F, V, V, M)
import SparseUnivariatePolynomialFunctions2(RF, F)
import FunctionSpaceUnivariatePolynomialFactor(R, F, UP)
import LinearOrdinaryDifferentialOperatorFactorizer(F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                           K, R, P, F)
```
upmp : (P, List K) -> P2
downmp : (P2, List K, List P) -> P
xpart : (F, SY) -> F
smpxpart : (P, SY, List K, List P) -> P
multint : (F, List F, SY) -> F
ulodo : (L, K) -> LQ
firstOrder : (F, F, F, SY) -> REC
rfSolve : (L, F, K, SY) -> U
ratlogsol : (LQ, List RF, K, SY) -> List F
expsois : (LQ, K, SY) -> List F
homosolve : (L, LQ, List RF, K, SY) -> List F
homosolve1 : (L, List F, K, SY) -> List F
norfl : (L, K, SY, N) -> List F
kovode : (LQ, K, SY) -> List F
doVarParams : (L, F, List F, SY) -> U
localmap : (F -> F, L) -> L
algSolve : (L, F, K, List K, SY) -> U
palgSolve : (L, F, K, K, SY) -> U
lastChance : (L, F, SY) -> U

diff := D()$L

smpxpart(p, x, l, lp) == downmp(primitivePart upmp(p, l), l, lp)
downmp(p, l, lp) == ground eval(p, l, lp)

-- left hand side has algebraic (not necessarily pure) coefficients
algSolve(op, g, k, l, x) ==
symbolIfCan(kx := ksec(k, l, x)) case SY => palgSolve(op, g, kx, k, x)
has?(operator kx, ALGOP) =>
rec := primitiveElement(kx::F, k::F)
z := rootOf(rec.prim)
lk:List K := [kx, k]
lv:List F := [(rec.pol1) z, (rec.pol2) z]
u := solve(localmap((f1:F):F +-> eval(f1, lk, lv), op), _
    eval(g, lk, lv), x))
    case "failed" => "failed"
rc := u::REC
kz := retract(z)@K
[eval(rc.particular, kz, rec.primelt),
    [eval(f, kz, rec.primelt) for f in rc.basis]]
lastChance(op, g, x)

doVarParams(eq, g, bas, x) ==
(u := particularSolution(eq, g, bas, (f1:F):F +-> int(f1, x)))
    case "failed" => lastChance(eq, g, x)
[u::F, bas]

lastChance(op, g, x) ==
```
-- one? degree op => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
(degree op) = 1 => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
"failed"

-- solves a0 y + a1 y' = g
-- does not check whether there is a solution in the field generated by
-- a0, a1 and g
firstOrder(a0, a1, g, x) ==
h := xpart(expint(- a0 / a1, x), x)
[h * int((g / h) / a1, x), [h]]

-- xpart(f,x) removes any constant not involving x from f
xpart(f, x) ==
l := reverse_! varselect(tower f, x)
lp := [k::P for k in l]
smpxpart(numer f, x, l, lp) / smpxpart(denom f, x, l, lp)

upmp(p, l) ==
empty? l => p::P2
up := univariate(p, k := first l)
l := rest l
ans:P2 := 0
while up ^= 0 repeat
    ans := ans + monomial(upmp(leadingCoefficient up, l), k, degree up)
    up := reductum up
ans

-- multint(a, [g1,...,gk], x) returns gk \int(g(k-1) \int(....g1 \int(a))...)
multint(a, l, x) ==
for g in l repeat a := g * xpart(int(a, x), x)
a

expsols(op, k, x) ==
-- one? degree op =>
(degree op) = 1 =>
firstOrder(multivariate(coefficient(op, 0), k),
multivariate(leadingCoefficient op, k), 0, x).basis
[xpart(expint(multivariate(h, k), x), x) for h in ricDsolve(op, ffactor)]

-- Finds solutions with rational logarithmic derivative
ratlogsol(op, sols, k, x) ==
bas := [xpart(multivariate(h, k), x) for h in sols]
degree(op) = #bas => bas -- all solutions are found already
rec := ReduceOrder(op, sols)
le := expsols(rec.eq, k, x)
int:List(F) := [xpart(multivariate(h, k), x) for h in rec.op]
concat_!( [xpart(multivariate(h, k), x) for h in sols],
           [multint(e, int, x) for e in le])

homosolve1(op, sols, k, x) ==
zero?(n := (degree(oper) - #sols)::N) => sols -- all solutions found
rec := ReduceOrder(oper, sols)
int:List(F) := [xpart(h, x) for h in rec.op]
concat_!(sols, [multint(e, int, x) for e in norf1(rec.eq, k, x, n::N)])

-- if the coefficients are rational functions, then the equation does not
-- have a proper 1st-order right factor over the rational functions
norf1(op, k, x, n) ==
  one? n => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
  (n = 1) => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
-- for order > 2, we check that the coeffs are still rational functions
symbolIfCan(kmax vark(coefficients op, x)) case SY =>
  eq := ulodo(op, k)
  n = 2 => kovode(eq, k, x)
  eq := last factor1 eq -- eq cannot have order 1
  degree(eq) = 2 =>
    empty?(bas := kovode(eq, k, x)) => empty()
    homosolve1(op, bas, k, x)
  empty()
  empty()

kovode(op, k, x) ==
  b := coefficient(op, 1)
  a := coefficient(op, 2)
  (u := kovacic(coefficient(op, 0), b, a, ffactor)) case "failed" => empty()
  p := map(z1+->multivariate(z1, k), u::UPUP)
  ba := multivariate(- b / a, k)
  -- if p has degree 2 (case 2), then it must be squarefree since the
  -- ode is irreducible over the rational functions, so the 2 roots of p
  -- are distinct and must yield 2 independent solutions.
  degree(p) = 2 => [xpart(expint(ba/(2::F) + e, x), x) for e in zerosOf p]
  -- otherwise take 1 root of p and find the 2nd solution by reduction of order
  [y1 := xpart(expint(ba / (2::F) + zeroOf p, x), x)]
  [y1, y1 * xpart(int(expint(ba, x) / y1**2, x), x)]

solve(op:L, g:F, x:SY) ==
  empty?(l := vark(coefficients op, x)) => constDsolve(op, g, x)
  symbolIfCan(k := kmax l) case SY => rfSolve(op, g, k, x)
  has?(operator k, ALGOP) => algSolve(op, g, k, l, x)
  lastChance(op, g, x)

ulodo(eq, k) ==
  op:LQ := 0
  while eq ^= 0 repeat
    op := op + monomial(univariate(leadingCoefficient eq, k), degree eq)
    eq := reductum eq
  op

-- left hand side has rational coefficients
rfSolve(eq, g, k, x) ==
op := ulodo(eq, k)
empty? remove_.!(k, varselect(kernels g, x)) => -- i.e. rhs is rational
rc := ratDsolve(op, univariate(g, k))
rc.particular case "failed" => -- this implies g ^= 0
doVarParams(eq, g, homosolve(eq, op, rc.basis, k, x), x)
[univariate(rc.particular::RF, k), homosolve(eq, op, rc.basis, k, x)]
doVarParams(eq, g, homosolve(eq, op, ratDsolve(op, 0).basis, k, x), x)

solve(op, g, x, a, y0) ==
(u := solve(op, g, x)) case "failed" => "failed"
hp := h := (u::REC).particular
b := (u::REC).basis
v::V := new(n := #y0, 0)
xK::K := kernel x
for i in minIndex v .. maxIndex v for yy in y0 repeat
  v.i := yy - eval(h, xK, a)
h := diff h
(sol := particularSolution(
  map_!((f1:F):F+->eval(f1,xK,a),wronskianMatrix(b,n)), v))
case "failed" => "failed"
for f in b for i in minIndex(s := sol::V) .. repeat
  hp := hp + s.i * f
hp

localmap(f, op) ==
ans::L := 0
while op ^= 0 repeat
  ans := ans + monomial(f leadingCoefficient op, degree op)
  op := reductum op
ans

-- left hand side has pure algebraic coefficients
palgSolve(op, g, xK, k, x) ==
rec := palgLODE(op, g, xK, k, x) -- finds solutions in the coef. field
rec.particular case "failed" =>
doVarParams(op, g, homosolve1(op, rec.basis, k, x), x)
[(rec.particular)::F, homosolve1(op, rec.basis, k, x)]

LODEEF dotabb

"LODEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODEEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"LODEEF" -> "ACFS"
package ODEEF ElementaryFunctionODESolver

--- ElementaryFunctionODESolver.input ---

)set break resume
)sys rm -f ElementaryFunctionODESolver.output
)spool ElementaryFunctionODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionODESolver
--E 1

)spool
)lisp (bye)

--- ElementaryFunctionODESolver.help ---

====================================================================
ElementaryFunctionODESolver examples
====================================================================

ElementaryFunctionODESolver provides the top-level functions for finding closed form solutions of ordinary differential equations and initial value problems.

See Also:
o )show ElementaryFunctionODESolver

---
ElementaryFunctionODESolver (ODEEF)

Exports:
solve

--- package ODEEF ElementaryFunctionODESolver ---

)abbrev package ODEEF ElementaryFunctionODESolver
++ Author: Manuel Bronstein
++ Date Created: 18 March 1991
++ Date Last Updated: 8 March 1994
++ Description:
++ \spad{ElementaryFunctionODESolver} provides the top-level
++ functions for finding closed form solutions of ordinary
++ differential equations and initial value problems.

ElementaryFunctionODESolver(R, F): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
         LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
         PrimitiveFunctionCategory)

N  ==> NonNegativeInteger
OP ==> BasicOperator
SY ==> Symbol
K  ==> Kernel F
EQ  ==> Equation F
V  ==> Vector F
M  ==> Matrix F
UP  ==> SparseUnivariatePolynomial F
P   ==> SparseMultivariatePolynomial(R, K)
LEQ ==> Record(left:UP, right:F)
NLQ ==> Record(dx:F, dy:F)
REC ==> Record(particular: F, basis: List F)
VEC ==> Record(particular: V, basis: List V)
ROW ==> Record(index: Integer, row: V, rh: F)
SYS ==> Record(mat:M, vec: V)
CHAPTER 6. CHAPTER E

U ==> Union(REC, F, "failed")
UU ==> Union(F, "failed")
OPDIFF ==> "%diff":::SY

Exports => with
solve: (M, V, SY) -> Union(VEC, "failed")
  ++ solve(m, v, x) returns \spad{[v_p, [v_1,...,v_m]]} such that
  ++ the solutions of the system \spad{D y = m y + v} are
  ++ \spad{v_p + c_1 v_1 + ... + c_m v_m} where the \spad{c_i's} are
  ++ constants, and the \spad{v_i's} form a basis for the solutions of
  ++ \spad{D y = m y}.
  ++ \spad{x} is the dependent variable.
solve: (M, SY) -> Union(List V, "failed")
  ++ solve(m, x) returns a basis for the solutions of \spad{D y = m y}.
  ++ \spad{x} is the dependent variable.
solve: (List EQ, List OP, SY) -> Union(VEC, "failed")
  ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
  ++ or, if the equations form a first order linear system, a solution
  ++ of the form \spad{[h_p, [b_1,...,b_n]]} where \spad{h_p} is a
  ++ particular solution and \spad{[b_1,...b_m]} are linearly independent
  ++ solutions of the associated homogeneous system.
  ++ error if the equations do not form a first order linear system
solve: (List F, List OP, SY) -> Union(VEC, "failed")
  ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
  ++ or, if the equations form a first order linear system, a solution
  ++ of the form \spad{[h_p, [b_1,...,b_n]]} where \spad{h_p} is a
  ++ particular solution and \spad{[b_1,...b_m]} are linearly independent
  ++ solutions of the associated homogeneous system.
  ++ error if the equations do not form a first order linear system
solve: (EQ, OP, SY) -> U
  ++ solve(eq, y, x) returns either a solution of the ordinary differential
  ++ equation \spad{eq} or "failed" if no non-trivial solution can be found;
  ++ If the equation is linear ordinary, a solution is of the form
  ++ \spad{h, [b_1,...,b_m]} where \spad{h} is a particular solution
  ++ and \spad{[b_1,...b_m]} are linearly independent solutions of the
  ++ associated homogeneous equation \spad{f(x,y) = 0};
  ++ A full basis for the solutions of the homogeneous equation
  ++ is not always returned, only the solutions which were found;
  ++ If the equation is of the form \spad{dy/dx = f(x,y)}, a solution is of
  ++ the form \spad{h(x,y)} where \spad{h(x,y) = c} is a first integral
  ++ of the equation for any constant \spad{c};
  ++ error if the equation is not one of those 2 forms;
solve: (F, OP, SY) -> U
  ++ solve(eq, y, x) returns either a solution of the ordinary differential
  ++ equation \spad{eq} or "failed" if no non-trivial solution can be found;
  ++ If the equation is linear ordinary, a solution is of the form
  ++ \spad{h, [b_1,...,b_m]} where \spad{h} is a particular solution and
  ++ and \spad{[b_1,...b_m]} are linearly independent solutions of the
  ++ associated homogeneous equation \spad{f(x,y) = 0};
  ++ A full basis for the solutions of the homogeneous equation
++ is not always returned, only the solutions which were found;
++ If the equation is of the form \(\frac{dy}{dx} = f(x,y)\), a solution is of
++ the form \(h(x,y)\) where \(h(x,y) = c\) is a first integral
++ of the equation for any constant \(c\);
solve: (EQ, OP, EQ, List F) -> UU
++ solve(eq, y, x = a, [y0,...,ym]) returns either the solution
++ of the initial value problem \(eq, y(a) = y0, y'(a) = y1,...\)
++ or "failed" if the solution cannot be found;
++ error if the equation is not one linear ordinary or of the form
++ \(\frac{dy}{dx} = f(x,y)\);
solve: (F, OP, EQ, List F) -> UU
++ solve(eq, y, x = a, [y0,...,ym]) returns either the solution
++ of the initial value problem \(eq, y(a) = y0, y'(a) = y1,...\)
++ or "failed" if the solution cannot be found;
++ error if the equation is not one linear ordinary or of the form
++ \(\frac{dy}{dx} = f(x,y)\);

Implementation ==> add
import ODEIntegration(R, F)
import IntegrationTools(R, F)
import NonLinearFirstOrderODESolver(R, F)

getfreelincoeff : (F, K, SY) -> F
getfreelincoeff1: (F, K, List F) -> F
getlincoeff : (F, K) -> F
getcoeff : (F, K) -> UU
parseODE : (F, OP, SY) -> Union(LEQ, NLQ)
parseLODE : (F, List K, UP, SY) -> LEQ
parseSYS : (List F, List OP, SY) -> Union(SYS, "failed")
parseSYSeq : (F, List K, List K, List F, SY) -> Union(ROW, "failed")
solve(diffeq:EQ, y:OP, x:SY) == solve(lhs diffeq - rhs diffeq, y, x)
solve(leq: List EQ, lop: List OP, x:SY) ==
    solve([lhs eq - rhs eq for eq in leq], lop, x)
solve(diffeq:EQ, y:OP, center:EQ, y0:List F) == solve(lhs diffeq - rhs diffeq, y, center, y0)
solve(m:M, x:SY) ==
    (u := solve(m, new(nrows m, 0), x)) case "failed" => "failed"
    u.basis
solve(m:M, v:V, x:SY) ==
    Lx := LinearOrdinaryDifferentialOperator(F, diff x)
    uu := solve(m, v, (z1,z2) -> solve(z1, z2, x)).
    $ElementaryFunctionLODESolver(R, F, Lx)$SystemODESolver(F, Lx)
    uu case "failed" => "failed"
    rec := uu::Record(particular: V, basis: M)
    [rec.particular, [column(rec.basis, i) for i in 1..ncols(rec.basis)]]
solve(diffeq:F, y:OP, center:EQ, y0:List F) ==
  a := rhs center
  kx:K := kernel(x := retract(lhs(center))@SY)
  (ur := parseODE(diffeq, y, x)) case NLQ =>
    -- not one?(#y0) => error "solve: more than one initial condition!"
    not (((#y0) = 1) => error "solve: more than one initial condition!"
    rc := ur::NLQ
    (u := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
    u::F - eval(u::F, [kx, retract(y(x::F))@K], [a, first y0])
  rec := ur::LEQ
  p := rec.left
  Lx := LinearOrdinaryDifferentialOperator(F, diff x)
  op:Lx := 0
  while p ^= 0 repeat
    op := op + monomial(leadingCoefficient p, degree p)
    p := reductum p
  solve(op, rec.right, x, a, y0)$ElementaryFunctionLODESolver(R, F, Lx)

solve(leq: List F, lop: List OP, x:SY) ==
  (u := parseSYS(leq, lop, x)) case SYS =>
    rec := u::SYS
    solve(rec.mat, rec.vec, x)
  error "solve: not a first order linear system"

solve(diffeq:F, y:OP, x:SY) ==
  (u := parseODE(diffeq, y, x)) case NLQ =>
    rc := u::NLQ
    (uu := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
    uu::F
  rec := u::LEQ
  p := rec.left
  Lx := LinearOrdinaryDifferentialOperator(F, diff x)
  op:Lx := 0
  while p ^= 0 repeat
    op := op + monomial(leadingCoefficient p, degree p)
    p := reductum p
    (uuu := solve(op, rec.right, x)$ElementaryFunctionLODESolver(R, F, Lx))
    case "failed" => "failed"
  uuu::REC

-- returns [M, v] s.t. the equations are D x = M x + v
parseSYS(eqs, ly, x) ==
  (n := #eqs) ^= #ly => "failed"
  m:M := new(n, n, 0)
  v:V := new(n, 0)
  xx := x::F
  if := [y xx for y in ly]
  lk0:List(K) := [retract(f)@K for f in if]
  lk1:List(K) := [retract(differentiate(f, x))@K for f in if]
for eq in eqs repeat
  (u := parseSYSeq(eq, lk0, lk1, lf, x)) case "failed" => return "failed"
  rec := u::ROW
  setRow_!(m, rec.index, rec.row)
  v(rec.index) := rec.rh
  [m, v]

parseSYSeq(eq, 10, 11, 1f, x) ==
  l := [k for k in varselect(kernels eq, x) | is?(k, OPDIFF)]
  empty? l or not empty? rest l or zero?(n := position(k := first l, l1)) =>
    "failed"
  c := getfreelincoeff1(eq, k, lf)
  eq := eq - c * k::F
  v:V := new(#l0, 0)
  for y in l0 for i in 1.. repeat
    ci := getfreelincoeff1(eq, y, lf)
    v.i := - ci / c
    eq := eq - ci * y::F
  [n, v, -eq]

-- returns either [p, g] where the equation (diffeq) is of the form p(D)(y) = g
-- or [p, q] such that the equation (diffeq) is of the form p dx + q dy = 0
parseODE(diffeq, y, x) ==
  f := y(x::F)
  l:List(K) := [retract(f)@K]
  n:N := 2
  for k in varselect(kernels diffeq, x) | is?(k, OPDIFF) repeat
    if (m := height k) > n then n := m
    n := (n - 2)::N
  -- build a list of kernels in the order [y^(n)(x),...,y''(x),y'(x),y(x)]
  for i in 1..n repeat
    l := concat(retract(f := differentiate(f, x))@K, l)
  k:K -- #$^# compiler requires this line and the next one too...
  c:F
  while not(empty? l) and zero?(c := getlincoeff(diffeq, k := first l))
    repeat l := rest l
  empty? l or empty? rest l => error "parseODE: equation has order 0"
  diffeq := diffeq - c * (k::F)
  ny := name y
  l := rest l
  height(k) > 3 => parseLODE(diffeq, l, monomial(c, #l), ny)
  (u := getcoeff(diffeq, k := first l)) case "failed" => [diffeq, c]
  eqrhs := (d := u::F) * (k::F) - diffeq
  freeOf?(eqrhs, ny) and freeOf?(c, ny) and freeOf?(d, ny) =>
    [monomial(c, 1) + d::UP, eqrhs]
  [diffeq, c]

-- returns [p, g] where the equation (diffeq) is of the form p(D)(y) = g
parseLODE(diffeq, l, p, y) ==
  not freeOf?(leadingCoefficient p, y) =>
error "parseLODE: not a linear ordinary differential equation"
d := degree(p)::Integer - 1
for k in l repeat
  p := p + monomial(c := getfreelincoeff(diffeq, k, y), d::N)
  d := d - 1
  diffeq := diffeq - c * (k::F)
freeOf?(diffeq, y) => [p, - diffeq]
error "parseLODE: not a linear ordinary differential equation"

getfreelincoeff(f, k, y) ==
  free0f?(c := getlincoeff(f, k), y) => c
  error "getfreelincoeff: not a linear ordinary differential equation"

getfreelincoeff1(f, k, ly) ==
  c := getlincoeff(f, k)
  for y in ly repeat
    not free0f?(c, y) =>
      error "getfreelincoeff: not a linear ordinary differential equation"
    c

getlincoeff(f, k) ==
  (u := getcoeff(f, k)) case "failed" =>
    error "getlincoeff: not an appropriate ordinary differential equation"
  u::F

getcoeff(f, k) ==
  (r := retractIfCan(univariate(denom f, k))@Union(P, "failed"))
  case "failed" or degree(p := univariate(numer f, k)) > 1 => "failed"
  coefficient(p, 1) / (r::P)

— ODEEF.dotabb —

"ODEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODEEF" -> "ACFS"

package SIGNEF ElementaryFunctionSign

— ElementaryFunctionSign.input —

)set break resume
This package provides functions to determine the sign of an elementary function around a point or infinity.

See Also:
o )show ElementaryFunctionSign

Exports:
sign
--- package SIGNEF ElementaryFunctionSign ---

)abbrev package SIGNEF ElementaryFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1989
++ Date Last Updated: 4 May 1992
++ Description:
++ This package provides functions to determine the sign of an
++ elementary function around a point or infinity.

ElementaryFunctionSign(R,F): Exports == Implementation where
  R : Join(IntegralDomain,OrderedSet,RetractableTo Integer,_
         LinearlyExplicitRingOver Integer,GcdDomain)
  F : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
         FunctionSpace R)

  N ==> NonNegativeInteger
  Z ==> Integer
  SY ==> Symbol
  RF ==> Fraction Polynomial R
  ORF ==> OrderedCompletion RF
  OFE ==> OrderedCompletion F
  K ==> Kernel F
  P ==> SparseMultivariatePolynomial(R, K)
  U ==> Union(Z, "failed")
  FS2 ==> FunctionSpaceFunctions2
  POSIT ==> "positive"
  NEGAT ==> "negative"

Exports ==> with
  sign: F -> U
    ++ sign(f) returns the sign of f if it is constant everywhere.
  sign: (F, SY, OFE) -> U
    ++ sign(f, x, a) returns the sign of f as x nears spad{a}, from both
    ++ sides if spad{a} is finite.
  sign: (F, SY, F, String) -> U
    ++ sign(f, x, a, s) returns the sign of f as x nears spad{a} from below
    ++ if s is "left", or above if s is "right".

Implementation ==> add
  import ToolsForSign R
  import RationalFunctionSign(R)
  import PowerSeriesLimitPackage(R, F)
  import TrigonometricManipulations(R, F)

  smpsign : P -> U
  sqfrSign: P -> U
  termSign: P -> U
  kerSign : K -> U
listSign: (List P, Z) -> U
insign : (F, SY, OFE, N) -> U
psign : (F, SY, F, String, N) -> U
ofesign : OFE -> U
overRF : OFE -> Union(ORF, "failed")

sign(f, x, a) ==
    not real? f => "failed"
    insign(f, x, a, 0)

sign(f, x, a, st) ==
    not real? f => "failed"
    psign(f, x, a, st, 0)

sign f ==
    not real? f => "failed"
    (u := retractIfCan(f)@Union(RF, "failed")) case RF => sign(u::RF)
    (un := smpsign numer f) case Z and (ud := smpsign denom f) case Z =>
        un::Z * ud::Z
    -- abort if there are any variables
    not empty? variables f => "failed"
    -- abort in the presence of algebraic numbers
    member?(coerce("rootOf")::Symbol,
        map(name, operators f)$ListFunctions2(BasicOperator, Symbol)) => "failed"
    -- In the last resort try interval evaluation where feasible.
    if R has ConvertibleTo Float then
        import Interval(Float)
        import Expression(Interval Float)
        mapfun : (R -> Interval(Float)) := z +-> interval(convert(z)$R)
        f2 : Expression(Interval Float) :=
            map(mapfun, f)$FS2(R, F, Interval(Float), Expression(Interval Float))
        r : Union(Interval(Float), "failed") := retractIfCan f2
        if r case "failed" then return "failed"
        negative? r => return(-1)
        positive? r => return 1
        zero? r => return 0
        "failed"
    "failed"

overRF a ==
    (n := whatInfinity a) = 0 =>
        (u := retractIfCan(retract(a)@OF)@Union(RF, "failed")) _
        case "failed" => "failed"
    u::RF::ORF::ORF
    n * plusInfinity()$ORF

ofesign a ==
    (n := whatInfinity a) ^= 0 => convert(n)@Z
    sign(retract(a)@OF)
insign(f, x, a, m) ==
  m > 10 => "failed" -- avoid infinite loops for now
  (uf := retractIfCan(f)@Union(RF,"failed")) case RF and
    (ua := overRF a) case ORF => sign(uf::RF, x, ua::ORF)
  eq : Equation OFE := equation(x :: F :: OFE,a)
  (u := limit(f,eq)) case "failed" => "failed"
  u case OFE =>
    (n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
    (v := retract(u::OFE)@F) = 0 =>
      (s := insign(differentiate(f, x), x, a, m + 1)) case "failed"
      => "failed"
      - s::Z * n
    sign v
  (u.leftHandLimit case "failed") or
    (u.rightHandLimit case "failed") => "failed"
  (ul := ofesign(u.leftHandLimit::OFE)) case "failed" => "failed"
  (ur := ofesign(u.rightHandLimit::OFE)) case "failed" => "failed"
  (ul::Z) = (ur::Z) => ul
  "failed"

psign(f, x, a, st, m) ==
  m > 10 => "failed" -- avoid infinite loops for now
  f = 0 => 0
  (uf := retractIfCan(f)@Union(RF,"failed")) case RF and
    (ua := retractIfCan(a)@Union(RF,"failed")) case RF =>
      sign(uf::RF, x, ua::RF, st)
  eq : Equation F := equation(x :: F,a)
  (u := limit(f,eq,st)) case "failed" => "failed"
  u case OFE =>
    (n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
    (v := retract(u::OFE)@F) = 0 =>
      (s := psign(differentiate(f,x),x,a,st,m + 1)) case "failed"
      => "failed"
      direction(st) * s::Z
    sign v
  (u.leftHandLimit case "failed") or
    (u.rightHandLimit case "failed") => "failed"
  (ul := ofesign(u.leftHandLimit::OFE)) case "failed" => "failed"
  (ur := ofesign(u.rightHandLimit::OFE)) case "failed" => "failed"
  (ul::Z) = (ur::Z) => ul
  "failed"

smrSign p ==
  (r := retractIfCan(p)@Union(R,"failed")) case R => sign(r::R)
  (u := sign(retract(unit(s := squareFree p))@R)) case "failed" =>
    "failed"
  ans := u::Z
  for term in factorList s | odd?(term.xpnt) repeat
    (u := sqfrSign(term.fctr)) case "failed" => return "failed"
  ans := ans * u::Z
  ans

sqfrSign p ==
  (u := termSign first(l := monomials p)) case "failed" => "failed"
  listSign(rest l, u::Z)
listSign(l, s) ==
  for term in l repeat
    (u := termSign term) case "failed" => return "failed"
    not(s = u::Z) => return "failed"
  s

termSign term ==
  (us := sign leadingCoefficient term) case "failed" => "failed"
  for var in (lv := variables term) repeat
    odd? degree(term, var) =>
      empty? rest lv and (vs := kerSign first lv) case Z =>
        return(us::Z * vs::Z)
    us::Z

kerSign k ==
  has?(op := operator k, "NEGAT") => -1
  has?(op, "POSIT") or is?(op, "pi"::SY) or is?(op,"exp"::SY) or
    is?(op,"cosh"::SY) or is?(op,"sech"::SY) => 1
  empty?(arg := argument k) => "failed"
  (s := sign first arg) case "failed" =>
    is?(op,"nthRoot" :: SY) =>
      even?(retract(second arg)@Z) => 1
    "failed"
  "failed"

is?(op,"log" :: SY) =>
  s::Z < 0 => "failed"
  sign(first arg - 1)

is?(op,"tanh" :: SY) or is?(op,"sinh" :: SY) or
  is?(op,"csch" :: SY) or is?(op,"coth" :: SY) => s

is?(op,"nthRoot" :: SY) =>
  even?(retract(second arg)@Z) =>
    s::Z < 0 => "failed"
  s

"failed"
package EFSTRUC ElementaryFunctionStructurePackage

— ElementaryFunctionStructurePackage.input —

)set break resume
)sys rm -f ElementaryFunctionStructurePackage.output
)spool ElementaryFunctionStructurePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionStructurePackage
--E 1

)spool
)lisp (bye)

——

— ElementaryFunctionStructurePackage.help —

====================================================================
ElementaryFunctionStructurePackage examples
====================================================================

ElementaryFunctionStructurePackage provides functions to test the algebraic independence of various elementary functions, using the Risch structure theorem (real and complex versions). It also provides transformations on elementary functions which are not considered simplifications.

See Also:
o )show ElementaryFunctionStructurePackage

——
ElementaryFunctionStructurePackage (EFSTRUC)

Exports:
normalize realElementary rootNormalize rischNormalize tanQ validExponential

--- package EFSTRUC ElementaryFunctionStructurePackage ---

)abbrev package EFSTRUC ElementaryFunctionStructurePackage
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 16 August 1995
++ Description:
++ ElementaryFunctionStructurePackage provides functions to test the
++ algebraic independence of various elementary functions, using the
++ Risch structure theorem (real and complex versions).
++ It also provides transformations on elementary functions
++ which are not considered simplifications.

ElementaryFunctionStructurePackage(R,F): Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

B ==> Boolean
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
SY ==> Symbol
K ==> Kernel F
UP ==> SparseUnivariatePolynomial F
SMP ==> SparseMultivariatePolynomial(R, K)
REC ==> Record(func:F, kers: List K, vals:List F)
U ==> Union(vec:Vector Q, func:F, fail: Boolean)
POWER ==> "%power":SY
NTHR ==> "nthRoot":SY
Exports ==> with
normalize: F -> F
  ++ normalize(f) rewrites \spad{f} using the least possible number of
  ++ real algebraically independent kernels.
normalize: (F, SY) -> F
  ++ normalize(f, x) rewrites \spad{f} using the least possible number of
  ++ real algebraically independent kernels involving \spad{x}.
rischNormalize: (F, SY) -> REC
  ++ rischNormalize(f, x) returns \spad{[g, [k1,...,kn], [h1,...,hn]]}
  ++ such that \spad{g = normalize(f, x)} and each \spad{ki} was
  ++ rewritten as \spad{hi} during the normalization.
realElementary: F -> F
  ++ realElementary(f) rewrites \spad{f} in terms of the 4 fundamental real
  ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
realElementary: (F, SY) -> F
  ++ realElementary(f,x) rewrites the kernels of \spad{f} involving
  ++ \spad{x} in terms of the 4 fundamental real
  ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
validExponential: (List K, F, SY) -> Union(F, "failed")
  ++ validExponential([k1,...,kn],f,x) returns \spad{g} if \spad{exp(f)=g}
  ++ and \spad{g} involves only \spad{k1...kn}, and "failed" otherwise.
rootNormalize: (F, K) -> F
  ++ rootNormalize(f, k) returns \spad{f} rewriting either \spad{k} which
  ++ must be an nth-root in terms of radicals already in \spad{f}, or some
  ++ radicals in \spad{f} in terms of \spad{k}.
tanQ: (Q, F) -> F
  ++ tanQ(q,a) is a local function with a conditional implementation.

Implementation ==> add
import TangentExpansions F
import IntegrationTools(R, F)
import IntegerLinearDependence F
import AlgebraicManipulations(R, F)
import InnerCommonDenominator(Z, Q, Vector Z, Vector Q)
k2Elem : (K, List SY) -> F
realElem : (F, List SY) -> F
smpElem : (SMP, List SY) -> F
deprel : (List K, K, SY) -> U
rootDep : (List K, K) -> U
qdeprel : (List K, K) -> U
factdeprel : (List K, K) -> U
toR : (List K, F) -> List K
toY : List K -> List F
toZ : List K -> List F
toU : List K -> List F
toV : List K -> List F
ktoY : K -> F
ktoZ : K -> F
ktoU : K -> F
ktoV : K -> F
gdCoef? : (Q, Vector Q) -> Boolean
goodCoef : (Vector Q, List K, SY) ->
    Union(Record(index:Z, ker:K), "failed")
tanRN : (Q, K) -> F
localnorm : F -> F
rooteval : (F, List K, K, Q) -> REC
logeval : (F, List K, K, Vector Q) -> REC
expeval : (F, List K, K, Vector Q) -> REC
taneval : (F, List K, K, Vector Q) -> REC
depeval : (F, List K, K, Vector Q) -> REC
expnosimp : (F, List K, K, Vector Q, List F, F) -> REC
tannosimp : (F, List K, K, Vector Q, List F, F) -> REC
rtNormalize : F -> F
rootNormalize0 : F -> REC
rootKernelNormalize: (F, List K, K) -> Union(REC, "failed")
tanSum : (F, List F) -> F

comb? := F has CombinatorialOpsCategory
mpiover2:F := pi()$F / (-2::F)

realElem(f, l) == smpElem(numer f, l) / smpElem(denom f, l)
realElementary(f, x) == realElem(f, [x])
realElementary f == realElem(f, variables f)
toY ker == [func for k in ker | (func := ktoY k) ^= 0]
toZ ker == [func for k in ker | (func := ktoZ k) ^= 0]
toU ker == [func for k in ker | (func := ktoU k) ^= 0]
toV ker == [func for k in ker | (func := ktoV k) ^= 0]
rtNormalize f == rootNormalize0(f).func
toR(ker, x) == select(s+->is?(s, NTHR) and first argument(s) = x, ker)

if R has GcdDomain then
tanQ(c, x) ==
    tanNa(rootSimp zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)
else
    tanQ(c, x) ==
    tanNa(zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)

-- tanSum(c, [a1,...,an]) returns f(c, a1,...,an) such that
-- if ai = tan/ui then f(c, a1,...,an) = tan(c + u1 + ... + un).
-- MUST BE CAREFUL FOR WHEN c IS AN ODD MULTIPLE of pi/2

tanSum(c, 1) ==
k := c / mpiover2 -- k = - 2 c / pi, check for odd integer
    -- tan((2n+1) pi/2 x) = - 1 / tan x
    (r := retractIfCan(k)@Union(Z, "failed")) case Z and odd?(r::Z) =>
        - inv tanSum 1
    tanSum concat(tan c, 1)

rootNormalize0 f ==
ker := select_!(s+->is?(s, NTHR) and empty? variables first argument s, tower f)$List(K)
empty? ker => [f, empty(), empty()]
(n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
for i in 1..n for kk in rest ker repeat
  (u := rootKernelNormalize(f, first(ker, i), kk)) case REC =>
    rec := u::REC
    rn := rootNormalize0(rec.func)
    return [rn.func, concat(rec.kers,rn.kers), concat(rec.vals, rn.vals)]
[f, empty(), empty()]
deprel(ker, k, x) ==
is?(k, "log"::SY) or is?(k, "exp"::SY) =>
  qdeprel([differentiate(g, x) for g in toY ker],
           differentiate(ktoY k, x))
is?(k, "atan"::SY) or is?(k, "tan"::SY) =>
  qdeprel([differentiate(g, x) for g in toU ker],
           differentiate(ktoU k, x))
is?(k, NTHR) => rootDep(ker, k)
comb? and is?(k, "factorial"::SY) =>
  factdeprel([x for x in ker | is?(x,"factorial"::SY) and x^=k],k)
  [true]
ktoY k ==
is?(k, "log"::SY) => k::F
is?(k, "exp"::SY) => first argument k
0
ktoZ k ==
is?(k, "log"::SY) => first argument k
is?(k, "exp"::SY) => k::F
0
ktoU k ==
is?(k, "atan"::SY) => k::F
is?(k, "tan"::SY) => first argument k
0
ktoV k ==
is?(k, "tan"::SY) => k::F
is?(k, "atan"::SY) => first argument k
0
smpElem(p, l) ==
  map(x+->k2Elem(x, l), y+->y::F, p)
  $PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)
k2Elem(k, l) ==
  ez, iez, tz2: F
  kf := k::F
not(empty? l) and empty? [v for v in variables kf | member?(v, l)] => kf
empty?(args :List F := [realElem(a, l) for a in argument k]) => kf
z := first args
is?(k, POWER) => (zero? z => 0; exp(last(args) * log z))
is?(k, "cot":SY) => inv tan z
is?(k, "acot":SY) => atan inv z
is?(k, "asin":SY) => atan(z / sqrt(1 - z**2))
is?(k, "acos":SY) => atan(sqrt(1 - z**2) / z)
is?(k, "asec":SY) => atan sqrt(1 - z**2)
is?(k, "acsc":SY) => atan inv sqrt(1 - z**2)
is?(k, "asinh":SY) => log(sqrt(1 + z**2) + z)
is?(k, "acosh":SY) => log(sqrt(z**2 - 1) + z)
is?(k, "atanh":SY) => log((z + 1) / (1 - z)) / (2::F)
is?(k, "acoth":SY) => log((z + 1) / (z - 1)) / (2::F)
is?(k, "asech":SY) => log((inv z) + sqrt(1 - z**2))
is?(k, "acsch":SY) => log((inv z) + sqrt(1 + inv(z**2)))
is?(k, "%paren":SY) or is?(k, "%box":SY) =>
  empty? rest args => z
kf
if has?(op := operator k, "htrig") then iez := inv(ez := exp z)
is?(k, "sinh":SY) => (ez - iez) / (2::F)
is?(k, "cosh":SY) => (ez + iez) / (2::F)
is?(k, "tanh":SY) => (ez - iez) / (ez + iez)
is?(k, "coth":SY) => (ez + iez) / (ez - iez)
is?(k, "sech":SY) => 2 * inv(ez + iez)
is?(k, "csch":SY) => 2 * inv(ez - iez)
if has?(op, "trig") then tz2 := tan(z / (2::F))
is?(k, "sin":SY) => 2 * tz2 / (1 + tz2**2)
is?(k, "cos":SY) => (1 - tz2**2) / (1 + tz2**2)
is?(k, "sec":SY) => (1 + tz2**2) / (1 - tz2**2)
is?(k, "csc":SY) => (1 + tz2**2) / (2 * tz2)
op args

--The next 5 functions are used by normalize, once a relation is found
depeval(f, lk, k, v) ==
is?(k, "log":SY) => logeval(f, lk, k, v)
is?(k, "exp":SY) => expeval(f, lk, k, v)
is?(k, "tan":SY) => taneval(f, lk, k, v)
is?(k, "atan":SY) => ataneval(f, lk, k, v)
is?(k, NTHR) => rooteval(f, lk, k, v(minIndex v))
[f, empty(), empty()]

rooteval(f, lk, k, n) ==
  nv := nthRoot(x := first argument k, m := retract(n)@Z)
  l := [r for r in concat(k, toR(lk, x)) |
    retract(second argument r)@Z ^= m]
  lv := [nv ** (n / (retract(second argument r)@Z::Q)) for r in l]
  [eval(f, l, lv), l, lv]

ataneval(f, lk, k, v) ==
w := first argument k
s := tanSum [tanQ(qelt(v, i), x) 
          for i in minIndex v .. maxIndex v for x in toV lk]
g := +/[qelt(v, i) * x for i in minIndex v .. maxIndex v for x in toU lk]
h:F := 
zero?(d := 1 + s * w) => mpiover2
atan((w - s) / d)
g := g + h
[eval(f, [k], [g]), [k], [g]]

gdCoef?(c, v) ==
for i in minIndex v .. maxIndex v repeat
  retractIfCan(qelt(v, i) / c) case "failed" =>
    return false
true
goodCoef(v, l, s) ==
for i in minIndex v .. maxIndex v for k in l repeat
  is?(k, s) and
  ((r := recip(qelt(v, i))) case Q) and
  (retractIfCan(r::Q) case Z)
  and gdCoef?(qelt(v, i), v) =>
    return([i, k])
"failed"
taneval(f, lk, k, v) ==
u := first argument k
fns := toU lk
c := u - +/[qelt(v, i) * x for i in minIndex v .. maxIndex v for x in fns]
(rec := goodCoef(v, lk, "tan"::SY)) case "failed" =>
tannosimp(f, lk, k, v, fns, c)
v0 := retract(inv qelt(v, rec.index)) case Z 
lv := [qelt(v, i) for i in minIndex v .. maxIndex v | i ^= rec.index]$List(Q)
l := [kk for kk in lk | kk ^= rec.ker]$List(Q)
g := tanSum(-v0 * c, concat(tanNa(k::F, v0), [
  [tanNa(x, - retract(a?v0)/Z) for a in lv for x in toV l]])
[eval(f, [rec.ker], [g]), [rec.ker], [g]]
tannosimp(f, lk, k, v, fns, c) ==
every?(x => is?(x, "tan"::SY), lk) =>
dd := (d := (cd := splitDenominator v).den) case F
newt := [tan(u / dd) for u in fns]$List(F)
newtan := [tanNa(t, d) for t in newt]$List(F)
h := tanSum(c, [tanNa(t, qelt(cd.num, i))
          for i in minIndex v .. maxIndex v for t in newt])
lk := concat(k, lk)
newtan := concat(h, newtan)
[eval(f, lk, newtan), lk, newtan]
h := tanSum(c, [tanQ(qelt(v, i), x)
          for i in minIndex v .. maxIndex v for x in toV lk])
exponsimp(f @= [exp(y / dd) for y in fns]$List(F)
new := newe ** d for e in newe$List(F)
newexp := h := */[e ** qelt(c.num, i)
  for i in minIndex v .. maxIndex v for e in newe] * g
lk := concat(lk, lk)
newexp := concat(h, newexp)
newexp := concat(h, newexp)
newexp := concat(h, newexp)

logeval(f @= first argument k
z := c := z / (*/[x**qelt(v, i)
  for x in toZ lk for i in minIndex v .. maxIndex v])
-- CHANGED log ktoZ x TO ktoY x SINCE WE WANT log exp f TO BE REPLACED BY f.
g := +/[qelt(v, i) * x
  for i in minIndex v .. maxIndex v for x in toY lk] + log c
newexp := [eval(f, [k], [h]), [k], [h]]

rischNormalize(f @= empty?(ker := varselect(tower f, v)) => [f, empty(), empty()]
  first(ker) ^= kernel(v)@K => error "Cannot happen"
ker := rest ker
(n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
for i in 1 .. n for kk in rest ker repeat
  klist := first(ker, i)
  -- NO EVALUATION ON AN EMPTY VECTOR, WILL CAUSE INFINITE LOOP
  (c := deprel(klist, kk, v)) case vec and not empty?(c.vec) =>
    rec := depeval(f, klist, kk, c.vec)
    n := rischNormalize(rec.func, v)
    return [n.func,
      concat(rec.kers, n.kers), concat(rec.vals, n.vals)]
  c case func =>
    n := rischNormalize(eval(f, [kk], [c.func])$List(F)
    return [n.func, concat(kk, n.kers), concat(c.func, n.vals)]
  [f, empty(), empty()]

rootNormalize(f @= (u := rootKernelNormalize(f, toR(tower f, first argument k), k))
  case "failed" => f
  (u::REC).func

rootKernelNormalize(f @= (c := rootDep(1, k)) case vec =>
rooteval(f, l, k, (c.vec)(minIndex(c.vec)))
"failed"

localnorm f ==
  for x in variables f repeat
    f := rischNormalize(f, x).func
f

validExponential(twr, eta, x) ==
  (c := solveLinearlyOverQ(construct([differentiate(g, x)
for g in (fns := toY twr)$List(F))@Vector(F),
differentiate(eta, x)]) case "failed" => "failed"
  v := c::Vector(Q)
g := eta - +/-[qelt(v, i) * yy
for i in minIndex v .. maxIndex v for yy in fns]
*/[exp(yy) ** qelt(v, i)
for i in minIndex v .. maxIndex v for yy in fns] * exp g

rootDep(ker, k) ==
  empty?(ker := toR(ker, first argument k)) => [true]
  [new(1,lcm(retract(second argument k)@Z,
  "lcm"/[retract(second argument r)@Z for r in ker])::Q)$Vector(Q)]

qdeprel(l, v) ==
  (u := solveLinearlyOverQ(construct(l)@Vector(F), v))
  case Vector(Q) => [u::Vector(Q)]
  [true]

expeval(f, lk, k, v) ==
  y := first argument k
  fns := toY lk
  g := y - +/-[qelt(v, i) * z for i in minIndex v .. maxIndex v for z in fns]
  (rec := goodCoef(v, lk, "exp"::SY)) case "failed" =>
    expnosimp(f, lk, k, vns, exp g)
  v0 := retract(inv qelt(v, rec.index))@Z
  lv := [qelt(v, i) for i in minIndex v .. maxIndex v | i ^= rec.index]$List(Q)
  l := [kk for kk in lk | kk ^= rec.ker]
  h :F := */[exp(z) ** (- retract(a * v0)@Z) for a in lv for z in toY l]
  h := h * exp(-v0 * g) * (k:F) ** v0
  [eval(f, [rec.ker], [h]), [rec.ker], [h]]

if F has CombinatorialOpsCategory then
  normalize f == rtNormalize localnorm factorials realElementary f

normalize(f, x) ==
  rtNormalize(rischNormalize(factorials(realElementary(f,x),x),x).func)

factdeprel(l, k) ==
  ((r := retractIfCan(n := first argument k)@Union(Z, "failed"))
case Z) and (r::Z > 0) => [factorial(r::Z)::F]
for x in l repeat
  m := first argument x
  ((r := retractIfCan(n - m)@Union(Z, "failed")) case Z) and
  (r::Z > 0) => return([*/[(m + i::F) for i in 1..r] * x::F])
[true]
else
  normalize f == rtNormalize localnorm realElementary f
  normalize(f, x) == rtNormalize(rischNormalize(realElementary(f,x),x).func)

— EFSTRUC.dotabb —

"EFSTRUC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFSTRUC"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EFSTRUC" -> "ACF"
"EFSTRUC" -> "FS"

package INTEF ElementaryIntegration

— ElementaryIntegration.input —

)set break resume
)sys rm -f ElementaryIntegration.output
)spool ElementaryIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryIntegration
--E 1

)spool
)lisp (bye)

— ElementaryIntegration.help —
This package provides functions for integration, limited integration, extended integration and the risch differential equation for elementary functions.

See Also:
- \texttt{\textbackslash show ElementaryIntegration}

---

\textbf{ElementaryIntegration (INTEF)}

\begin{figure}
\begin{center}
\begin{tikzpicture}
\node (INTEF) at (0,0) {INTEF};
\node (ACF) at (-2,-2) {ACF};
\node (FS) at (2,-2) {FS};
\draw[->] (INTEF) -- (ACF);
\draw[->] (INTEF) -- (FS);
\end{tikzpicture}
\end{center}
\end{figure}

\textbf{Exports:}
- \texttt{lfextendedint}
- \texttt{lfextlimint}
- \texttt{lfintfieldint}
- \texttt{lfintegrate}
- \texttt{lflimitedint}

---

\texttt{\textbackslash abbrev package INTEF ElementaryIntegration}

++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 24 October 1995
++ Description:
++ This package provides functions for integration, limited integration, extended integration and the risch differential equation for elementary functions.

\texttt{ElementaryIntegration(R, F): Exports == Implementation where}
\begin{verbatim}
R : Join(GcdDomain, OrderedSet, CharacteristicZero, RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
\end{verbatim}
FunctionSpace R)

SE  ==> Symbol
K   ==> Kernel F
P   ==> SparseMultivariatePolynomial(R, K)
UP  ==> SparseUnivariatePolynomial F
RF  ==> Fraction UP
IR  ==> IntegrationResult F
FF  ==> Record(ratpart:RF, coeff:RF)
LLG ==> List Record(coeff:F, logand:F)
U2  ==> Union(Record(ratpart:F, coeff:F), "failed")
U3  ==> Union(Record(mainpart:F, limitedlogs:LLG), "failed")
ANS ==> Record(special:F, integrand:F)
PSOL == Record(ans:F, right:F, sol?:Boolean)
FAIL == error "failed - cannot handle that integrand"
ALGOP == "%alg"
OPDIFF == "%diff":SE

Exports == with
  lfxextendedint: (F, SE, F) -> U2
    ++ lfxextendedint(f, x, g) returns functions \spad{[h, c]} such that
    ++ \spad{dh/dx = f - cg}, if (h, c) exist, "failed" otherwise.
  lflimitedint : (F, SE, List F) -> U3
    ++ lflimitedint(f,x,[g1,...,gn]) returns functions \spad{[h,[[ci, gi]]]}
    ++ such that the gi's are among \spad{[g1,...,gn]}, and
    ++ \spad{d(h+sum(ci log(gi)))/dx = f}, if possible, "failed" otherwise.
  lfinfieldint : (F, SE) -> Union(F, "failed")
    ++ lfinfieldint(f, x) returns a function g such that \spad{dg/dx = f}
    ++ if g exists, "failed" otherwise.
  lfinegrate : (F, SE) -> IR
    ++ lfinegrate(f, x) = g such that \spad{dg/dx = f}.
  lfxextlimint : (F, SE, K, List K) -> U2
    ++ lfxextlimint(f,x,k,[k1,...,kn]) returns functions \spad{[h, c]}
    ++ such that \spad{dh/dx = f - c dk/dx}. Value h is looked for in a field
    ++ containing f and k1,...,kn (the ki's must be logs).

Implementation == add
  import IntegrationTools(R, F)
  import ElementaryRischDE(R, F)
  import RationalIntegration(F, UP)
  import AlgebraicIntegration(R, F)
  import AlgebraicManipulations(R, F)
  import ElementaryRischDESystem(R, F)
  import TranscendentalIntegration(F, UP)
  import PureAlgebraicIntegration(R, F)
  import IntegrationResultFunctions2(F, F)
  import IntegrationResultFunctions2(RF, F)
  import FunctionSpacePrimitiveElement(R, F)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                K, R, P, F)
algfint : (F, K, List K, SE) -> IR
algfextint : (F, K, List K, SE, F) -> U2
algflimint : (F, K, List K, SE, List F) -> U3
primextint : (F, SE, K, F) -> U2
expextint : (F, SE, K, F) -> U2
primlimint : (F, SE, K, List F) -> U3
explimint : (F, SE, K, List F) -> U3
algprimint : (F, K, K, SE) -> IR
algexpint : (F, K, K, SE) -> IR
primint : (F, SE, K) -> IR
expint : (F, SE, K) -> IR
tanint : (F, SE, K) -> IR
prim? : (K, SE) -> Boolean
isx? : (F, SE) -> Boolean
addx : (IR, F) -> IR
cfind : (F, LLG) -> F
lfintegrate0: (F, SE) -> IR
unknownint : (F, SE) -> IR
unkextint : (F, SE, F) -> U2
unklimint : (F, SE, List F) -> U3
tryChangeVar: (F, K, SE) -> Union(IR, "failed")
droponex : (F, F, K, F) -> Union(F, "failed")

prim?(k, x) == is?(k, "log":SE) or has?(operator k, "prim")

primint(f, x, k) ==
eta' := differentiate(eta := first argument k, x)
r1 :=
tanintegrate(univariate(f, k),
(x1:UP):UP +-> differentiate(x1,
(x2:F):F +-> differentiate(x2, x),
monomial(eta', 2) + eta':UP),
(x3:Integer,x4:F,x5:F):Union(List F,"failed") +->
rischDEsys(x3, 2 * eta, x4, x5, x,
(x6:F,x7:List F):U3 +-> lflimitedint(x6, x, x7),
(x8:F,x9:F):U2 +-> lfextendedint(x8, x, x9)))
map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

-- tries various tricks since the integrand contains something not elementary
unknownint(f, x) ==
((r := retractIfCan(f)@Union(K, "failed")) case K) and
is?(k := r::K, OPDIFF) and
((ka:=retractIfCan(a:=second(l:=argument k))@Union(K,"failed"))case K) and
((z := retractIfCan(zz := third l)@Union(SE, "failed")) case SE) and
(z::SE = x) and ((u := droponex(first l, a, ka, zz)) case F) => u::F::IR
da := differentiate(a := denom(f)::F, x)) "= 0 and
zero? differentiate(c := numer(f)::F / da, x) => (c * log a)::IR
mkAnswer(0, empty(), [[f, x::F]])
droponex(f, a, ka, x) ==
  (r := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
  is?(op := operator(k := r::K), OPDIFF) =>
    (z := third(arg := argument k)) = a => op [first arg, second arg, x]
    (u := droponex(first arg, a, ka, x)) case "failed" => "failed"
    op [u::F, second arg, z]
  eval(f, [ka], [x])

unklimint(f, x, lu) ==
  for u in lu | u ^= 0 repeat
    zero? differentiate(c := f * u / differentiate(u, x), x) => [0, [[c,u]]]
  "failed"

unkextint(f, x, g) ==
  zero?(g' := differentiate(g, x)) => "failed"
  zero? differentiate(c := f / g', x) => [0, c]
  "failed"

isx?(f, x) ==
  (k := retractIfCan(f)@Union(K, "failed")) case "failed" => false
  (r := symbolIfCan(k::K)) case "failed" => false
  r::SE = x

alglfint(f, k, l, x) ==
  xf := x::F
  symbolIfCan(kx := ksec(k,l,x)) case SE => addx(palgint(f, kx, k), xf)
  is?(kx, "exp"::SE) => addx(algexpint(f, kx, k, x), xf)
  prim?(kx, x) => addx(algprimint(f, kx, k, x), xf)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y := rootOf(rec.prim)
    map((x1:F):F +-> eval(x1, retract(y)@K, rec.primelt),
        ifintegrate(eval(f, [kx,k], [(rec.pol1) y, (rec.pol2) y]), x))
  unknownint(f, x)

alglfextint(f, k, l, x, g) ==
  symbolIfCan(kk := ksec(k,l,x)) case SE => palgextint(f, kk, k, g)
  has?(operator kk, ALGOP) =>
    rec := primitiveElement(kk::F, k::F)
    y := rootOf(rec.prim)
    lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
    (u := ifextendedint(eval(f, [kx,k], lrhs), x,
                         eval(g, [kx,k], lrhs))) case "failed" => "failed"
    ky := retract(y)@K
    r := u::Record(ratpart:F, coeff:F)
    [eval(r.ratpart,ky,rec.primelt), eval(r.coeff,ky,rec.primelt)]
  is?(kx, "exp"::SE) or is?(kx, "log"::SE) => FAIL
  unkextint(f, x, g)
algflimit(f, k, 1, x, lu) ==
symbolIfCan(kx := ksec(k, 1, x)) case SE => palglimint(f, kx, k, lu)
has?(operator kx, ALGOP) =>
rec := primitiveElement(kx::F, k::F)
y := rootOf(rec.prim)
lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
(u := lflimitedint(eval(f, [kx, k], lrhs), x, map((x1:F):F +-> eval(x1, [kx, k], lrhs), lu))) case "failed" => "failed"
ky := retract(y)@K
r := u::Record(mainpart:F, limitedlogs:LLG)
[eval(r.mainpart, ky, rec.prime),
[eval(rc.coeff, ky, rec.prime),
eval(rc.logand, ky, rec.prime)] for rc in r.limitedlogs]
is?(kx, "exp":SE) or is?(kx, "log":SE) => FAIL
unklimint(f, x, lu)

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
and F has Join(LiouvilleFunctionCategory, RetractableTo SE) then
import PatternMatchIntegration(R, F)
lfintegrate(f, x) == intPatternMatch(f, x, lfintegrate0, pmintegrate)
else lfintegrate(f, x) == lfintegrate0(f, x)

lfintegrate0(f, x) ==
zero? f => 0
xf := x::F
empty?(l := varselect(kernels f, x)) => (xf * f)::IR
symbolIfCan(k := kmax l) case SE =>
map((x1:RF):F +-> multivariate(x1, k), integrate univariate(f, k))
is?(k, "tan":SE) => addx(tanint(f, x, k), xf)
prim?(k, x) => addx(primint(f, x, k), xf)
has?(operator k, ALGOP) => algflint(f, k, l, x)
unknownint(f, x)

addx(i, x) ==
elem? i => i
mkAnswer(ratpart i, logpart i,
[[ne.integrand, x] for ne in notelem i])

tryChangeVar(f, t, x) ==
z := new()$Symbol
g := subst(f / differentiate(t::F, x), [t], [z::F])
freeOf?(g, x) => -- can we do change of variables?
map((x1:F):F +-> eval(x1, kernel z, t::F), lfinetrate(g, z))
"failed"
algexpint(f, t, y, x) ==
(u := tryChangeVar(f, t, x)) case IR => u::IR
algint(f, t, y,
Bug #100 is an infinite loop that eventually kills Axiom from the input

\texttt{integrate((z}^{a+1})^b,z)\]

Line 2 of this function used to read:

\texttt{symbolIfCan(k := kmax(1 := union(1, varselect(kernels g, x))))}

The loop occurs when the call to union causes

\begin{verbatim}
  a log(z) %e
\end{verbatim}

to get added to the list every time. This gives the argument to kmax

\begin{verbatim}
  a log(z) arg1= [z,%e ]
\end{verbatim}

and the result being

\begin{verbatim}
  a log(z) %e
\end{verbatim}

We keep coming back to process this term, which ends up putting the same term back on the list and we loop. Waldek's solution is to remove the union call.

The original patch fixed the infinite regression mentioned above but caused Axiom to return a closed form of the integral:

\texttt{integrate(asech(x)/x,x)}

which should not have a closed form. This is referenced in the FriCAS SVN revision 279.

Essentially this new patch uses only logarithms of rational functions when integrating rational functions. It is unclear whether this is the correct fix.
lfextendedint(f, x, g) ==
  empty?(l := varselect(kernels f, x)) => [x::F * f, 0]
  symbolIfCan(k := kmax(l))
  case SE =>
    g1 :=
      empty?(l1 := varselect(kernels g, x)) => 0::F
      kmax(l1) = k => g
      0::F
      map((x1:RF):F +-> multivariate(x1, k),
          extendedint(univariate(f, k),
                      univariate(g1, k)))
    is?(k, "exp"::SE) => expextint(f, x, k, g)
    prim?(k, x) => primextint(f, x, k, g)
    has?(operator k, ALGOP) => alglfextint(f, k, l, x, g)
  unkextint(f, x, g)

This is part of the fix for bug 100. Line 2 of this function used to read:
  symbolIfCan(k := kmax(l := union(l, vark(lu, x)))) case SE =>
See the above discussion for why this causes an infinite loop.

lflimitedint(f, x, lu) ==
  empty?(l := varselect(kernels f, x)) => [x::F * f, empty()]
  symbolIfCan(k := kmax(l)) case SE =>
    map((x1:RF):F +-> multivariate(x1, k),
        limitedint(univariate(f, k),
                     [univariate(u, k) for u in lu]))
    is?(k, "exp"::SE) => explimint(f, x, k, lu)
    prim?(k, x) => primlimint(f, x, k, lu)
    has?(operator k, ALGOP) => alglfint(f, k, l, x, lu)
    unklimint(f, x, lu)

lfinfieldint(f, x) ==
  (u := lfextendedint(f, x, 0)) case "failed" => "failed"
  u.ratpart

primextint(f, x, k, g) ==
  lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
  (u1 := primextendedint(univariate(f, k),
                         (x1:UP):UP +-> differentiate(x1,
                         (x2:F):F +-> differentiate(x2, x),
                         differentiate(k::F, x)::UP),
                         (x3:F):U2 +-> lfextlimint(x3, x, k, lk),
                         univariate(g, k))) case "failed"
    => "failed"
  u1 case FF =>
    [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
expextint(f, x, k, g) ==
(u1 := expextendedint(univariate(f, k),
(x1:UP):UP +-> differentiate(x1,
(x2:F):F ++-> differentiate(x2, x),
monomial(differentiate(first argument k, x), 1)),
(x3:Integer,x4:F):PSOL++->rischDE(x3, first argument k, x4, x,
(x5:F,x6:List F):U3 ++-> iflimitedint(x5, x, x6),
(x7:F,x8:F):U2++->lfextendedint(x7, x, x8)), univariate(g, k))
  case "failed" => "failed"
  u1 case FF =>
    [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
(u2 := lfextendedint(u1.a0, x, g)) case "failed" => "failed"
  [multivariate(u1.answer, k) + u2.ratpart, u2.coeff]

primint(f, x, k) ==
lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
r1 := primintegrate(univariate(f, k),
  (x1:UP):UP +-> differentiate(x1,
  (x2:F):F ++-> differentiate(x2, x),
  differentiate(k::F, x)::UP),
  (x3:F):U2 +-> lfextlimint(x3, x, k, lk))
  map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

lfextlimint(f, x, k, lk) ==
  not((u1 := lfextendedint(f, x, differentiate(k::F, x)))
  case "failed") => u1
twr := tower f
  empty?(lg := [kk for kk in lk | not member?(kk, twr)]) => "failed"
  is?(k, "log"::SE) =>
    (u2 := iflimitedint(f, x,
        [first argument u for u in union(lg, [k])]) case "failed"
        => "failed"
    cf := cfind(first argument k, u2.limitedlogs)
    u2.mainpart - cf * k::F +
        +/[c.coeff * log(c.logand) for c in u2.limitedlogs], cf]
  "failed"

  cfind(f, l) ==
    for u in l repeat
      f = u.logand => return u.coeff
      0

expint(f, x, k) ==
  eta := first argument k
  r1 :=
    expintegrate(univariate(f, k),
    (x1:UP):UP +-> differentiate(x1,
    (x2:F):F ++-> differentiate(x2, x),
  ...
monomial(differentiate(eta, x), 1)),
(x3:Integer,x4:F):PSOL+->rischDE(x3, eta, x4, x,
(x5:F,x6:List F):U3 +-> 1flimitedint(x5, x, x6),
(x7:F,x8:F):U2++->1fextendedint(x7, x, x8))
map((x1:RF):F+->multivariate(x1, k), r1.answer) + 1fintegrate(r1.a0, x)

primlimint(f, x, k, lu) ==
lk := vare select([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
(u1 :=
primlimitedint(univariate(f, k),
(x1:UP):UP++->differentiate(x1,
(x2:F):F++->differentiate(x2, x), differentiate(k::F, x)::UP),
(x3:F):U2++->1fextlimint(x3,x,k,lk),
[univariate(u, k) for u in lu])
case "failed" => "failed"
l := [[multivariate(lg.coeff, k),multivariate(lg.logand, k)]
for lg in u1.answer.limitedlogs]$LLG
(u2 := 1flimitedint(u1.a0, x, lu)) case "failed" => "failed"
[multivariate(u1.answer.mainpart, k) + u2.mainpart,
concat(u2.limitedlogs, l)]

explimint(f, x, k, lu) ==
eta := first argument k
(u1 :=
explimitedint(univariate(f, k),
(x1:UP):UP++->differentiate(x1,
(x2:F):F++->differentiate(x2, x), monomial(differentiate(eta,x), 1)),
(x3:Integer,x4:F):PSOL+->rischDE(x3, eta, x4, x,
(x5:F,x6:List F):U3 +-> 1flimitedint(x5, x, x6),
(x7:F,x8:F):U2++->1fextendedint(x7, x, x8)),
[univariate(u, k) for u in lu]) case "failed" => "failed"
l := [[multivariate(lg.coeff, k),multivariate(lg.logand, k)]
for lg in u1.answer.limitedlogs]$LLG
(u2 := 1flimitedint(u1.a0, x, lu)) case "failed" => "failed"
[multivariate(u1.answer.mainpart, k) + u2.mainpart,
concat(u2.limitedlogs, l)]

— INTEF.dotabb —

"INTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTEF" -> "ACF"
"INTEF" -> "FS"
package RDEEF ElementaryRischDE

--- ElementaryRischDE.input ---

)set break resume
)sys rm -f ElementaryRischDE.output
)spool ElementaryRischDE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryRischDE
--E 1

)spool
)lisp (bye)

---

--- ElementaryRischDE.help ---

=======================================
ElementaryRischDE examples
=======================================

Risch differential equation, elementary case.

See Also:
  o )show ElementaryRischDE

---
ElementaryRischDE (RDEEF)

Exports:
rischDE

— package RDEEF ElementaryRischDE —

)abbrev package RDEEF ElementaryRischDE
++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 2 November 1995
++ Description:
++ Risch differential equation, elementary case.

ElementaryRischDE(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
    RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, AlgebraicallyClosedField,
    FunctionSpace R)

N ==> NonNegativeInteger
Z ==> Integer
SE ==> Symbol
LF ==> List F
K ==> Kernel F
LK ==> List K
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
GP ==> LaurentPolynomial(F, UP)
Data ==> List Record(coeff:Z, argument:P)
RRF ==> Record(mainpart:F, limitedlogs:List NL)
NL ==> Record(coeff:F, logand:F)
U ==> Union(RRF, "failed")
UF ==> Union(F, "failed")
UUP ==> Union(UP, "failed")
UGP ==> Union(GP, "failed")
URF ==> Union(RF, "failed")
UEX ==> Union(Record(ratpart:F, coeff:F), "failed")
PSOL==> Record(ans:F, right:F, sol?:Boolean)
FAIL==> error("Function not supported by Risch d.e.")
ALGOP ==> "%alg"

Exports ==> with
rischDE: (Z, F, F, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
  ++ rischDE(n, f, g, x, lim, ext) returns \spad{[y, h, b]} such that
  ++ \spad{dy/dx + n df/dx y = h} and \spad{b := h = g}.
  ++ The equation \spad{dy/dx + n df/dx y = g} has no solution
  ++ if \spad{h \nless g} \ (y is a partial solution in that case).
  ++ Notes: lim is a limited integration function, and
  ++ ext is an extended integration function.

Implementation ==> add
import IntegrationTools(R, F)
import TranscendentalRischDE(F, UP)
import TranscendentalIntegration(F, UP)
import PureAlgebraicIntegration(R, F)
import FunctionSpacePrimitiveElement(R, F)
import ElementaryFunctionStructurePackage(R, F)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, 
  K, R, P, F)

RF2GP: RF -> GP
makeData : (F, SE, K) -> Data
normal10 : (Z, F, F, SE) -> UF
normalise0: (Z, F, F, SE) -> PSOL
normalise : (Z, F, F, SE, K, (F, LF) -> U, (F, F) -> UEX) -> PSOL
rischDEalg: (Z, F, F, F, K, LK, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
rischDElog: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
rischDEexp: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
boundAt0 : (LK, F, Z, Z, SE, K, (F, LF) -> U) -> Z
boundInf : (LK, F, Z, Z, SE, K, (F, LF) -> U) -> Z
logdegrad : (LK, F, UP, Z, SE, K,(F,LF)->U, (F,F) -> UEX) -> UUP
expdegrad : (LK, F, UP, Z, SE, K,(F,LF)->U, (F,F) -> UEX) -> UUP
logdeg : (UP, F, Z, SE, F, (F, LF) -> U, (F, F) -> UEX) -> UUP
expdeg : (UP, F, Z, SE, F, (F, LF) -> U, (F, F) -> UEX) -> UUP
exppolyint: (UP, (Z, F) -> PSOL) -> UUP
RRF2F : RRF -> F
logdiff : (List K, List K) -> List K

RF2GP f == (numer(f)::GP exquo denom(f)::GP)::GP
CHAPTER 6. CHAPTER E

logdiff(twr, bad) ==
[u for u in twr | is?(u, "log"::SE) and not member?(u, bad)]

rischDEalg(n, nfp, f, g, k, l, x, limint, extint) ==
symbolIfCan(kx := ksec(k, l, x)) case SE =>
(u := palgRDE(nfp, f, g, kx, k,
  (z1,z2,z3) +-> normal0(n, z1, z2, z3))) case "failed"
  => [0, 0, false]
[u::F, g, true]
has?(operator kx, ALGOP) =>
  rec := primitiveElement(kx::F, k::F)
  y := rootOf(rec.prim)
lk:LK := [kx, k]
lv:LF := [(rec.pol1) y, (rec.pol2) y]
rc := rischDE(n, eval(f, lk, lv), eval(g, lk, lv), x, limint, extint)
rc.sol? => [eval(rc.ans, retract(y)@K, rec.primeIt), rc.right, true]
[0, 0, false]
FAIL

-- solve y' + n f' y = g for a rational function y
rischDE(n, f, g, x, limitedint, extendedint) ==
zero? g => [0, g, true]
zero?(nfp := n * differentiate(f, x)) =>
  (u := limitedint(g, empty())) case "failed" => [0, 0, false]
  [u.mainpart, g, true]
freeOf?(y := g / nfp, x) => [y, g, true]
v1 := varselect(union(kernels nfp, kernels g), x)
symbolIfCan(k := kmax v1) case SE => normalise0(n, f, g, x)
is?(k, "log"::SE) or is?(k, "exp"::SE) =>
  normalise(n, nfp, f, g, k, x, limitedint, extendedint)
has?(operator k, ALGOP) =>
  rischDEalg(n, nfp, f, g, k, v1, x, limitedint, extendedint)
FAIL

normal0(n, f, g, x) ==
rec := normalise0(n, f, g, x)
rec.sol? => rec.ans
"failed"

-- solve y' + n f' y = g
-- when f' and g are rational functions over a constant field
normalise0(n, f, g, x) ==
k := kernel(x)@K
if (data1 := search(f, tab)) case "failed" then
tab.f := data := makeData(f, x, k)
else data := data1::Data
f' := nfprime := n * differentiate(f, x)
p:P := 1
for v in data | (m := n * v.coeff) > 0 repeat
  p := p * v.argument ** (m::N)
\[ f' := f' - m \cdot \frac{\text{differentiate}(v.\text{argument}::F, x)}{v.\text{argument}::F} \]
\[ \text{rec} := \text{baseRDE}(\text{univariate}(f', k), \text{univariate}(p::F * g, k)) \]
\[ y := \frac{\text{multivariate}(\text{rec.ans}, k)}{p::F} \]
\[ \text{rec.nosol} \Rightarrow [y, \text{differentiate}(y, x) + \text{nfp} \cdot y, \text{false}] \]
\[ [y, g, \text{true}] \]

-- make \( f \) weakly normalized, and solve \( y' + n \cdot f' \cdot y = g \)
\text{normalise}(n, \text{nfp}, f, g, x, k, \text{limitedint}, \text{extendedint}) ==

\begin{enumerate}
  \item if (\text{data1} := \text{search}(f, \text{tab})) case "failed" then
    \[ \text{tab.f} := \text{data} := \text{makeData}(f, x, k) \]
  \item else \text{data} := \text{data1::Data}
  \end{enumerate}
\begin{enumerate}
  \item \[ p := 1 \]
  \item for \( v \) in \text{data} | (m := n \cdot v.\text{coeff}) > 0 \text{ repeat}
    \[ p := p \cdot v.\text{argument}^{m::N} \]
  \item \[ f := f - v.\text{coeff} \cdot \log(v.\text{argument}::F) \]
  \item \[ \text{nfp} := \text{nfp} - m \cdot \frac{\text{differentiate}(v.\text{argument}::F, x)}{v.\text{argument}::F} \]
  \item \[ \text{newf} := \text{univariate}(\text{nfp}, k) \]
  \item \[ \text{newg} := \text{univariate}(p::F * g, k) \]
  \item \[ \text{twr} := \text{union}((\text{logdiff}(\text{tower } f, \text{empty}()), \text{logdiff}(\text{tower } g, \text{empty}()))) \]
\end{enumerate}
\[ \text{ans1 :=} \]
\begin{enumerate}
  \item \[ \text{is?(k, "log"::SE) =>} \]
    \text{rischDElog}(\text{twr}, \text{newf}, \text{newg}, x, k,
    \[ z1 \mapsto \text{differentiate}(z1, (z2::F):F \mapsto \text{differentiate}(z2, x), \]
    \text{differentiate}(k::F, x)::UP),
    \text{limitedint}, \text{extendedint}) \]
  \item \[ \text{is?(k, "exp"::SE) =>} \]
    \text{rischDEexp}(\text{twr}, \text{newf}, \text{newg}, x, k,
    \[ z1 \mapsto \text{differentiate}(z1, (z2::F):F \mapsto \text{differentiate}(z2, x), \]
    \text{monomial}(\text{differentiate}(\text{first argument } k, x), 1)),
    \text{limitedint}, \text{extendedint}) \]
\end{enumerate}
\[ \text{ans1 case "failed" => [0, 0, \text{false}] }\]
\[ [\text{multivariate}(\text{ans1::RF}, k) / p::F, g, \text{true}] \]

-- find the \( n \cdot \log(P) \) appearing in \( f \), where \( P \) is in \( P \), \( n \) in \( Z \)
\text{makeData}(f, x, k) ==
\begin{enumerate}
  \item \[ \text{disasters} := \text{empty()}$\text{Data} \]
  \item \[ \text{fnum} := \text{numerator } f \]
  \item \[ \text{fden} := \text{denominator } f \]
  \item for \( u \) in \text{varselect(kernels } f, x) | \text{is?}(u, "log"::SE) \text{ repeat}
    \text{logand} := \text{first argument } u
    \[ \text{if zero?(degree } \text{univariate}(\text{fden}, u)) \text{ and}
      \text{one?(degree } \text{numerator } \text{univariate}(\text{fnum}, u)) \text{ then}
      \text{cf} := (\text{leadingCoefficient } \text{numerator } \text{univariate}(\text{fnum}, u)) / \text{fden}
      \text{if } (\text{n} := (\text{retractIfCan(cf)}@\text{Union}(Z, "failed"))) \text{ case } Z \text{ then}
        \text{disasters} := \text{concat}([\text{n::Z}, \text{numerator } \text{logand}], \text{disasters})
      \text{if degree}(\text{denominator } \text{logand}, k) > 0 \text{ then}
        \text{disasters} := \text{concat}([-\text{n::Z}, \text{denominator } \text{logand}], \text{disasters})
      \text{disasters} \]
      \[ \text{else}
      \text{disasters} := \text{concat}([-\text{n::Z}, \text{denominator } \text{logand}], \text{disasters})
    \]
rischDElog(twr, f, g, x, theta, driv, limint, extint) ==
(u := monomRDE(f, g, driv)) case "failed" => "failed"
(v := polyDElog(twr, u.a, retract(u.b), retract(u.c), x, theta, driv,
   limint, extint)) case "failed" => "failed"
v::UP / u.t

rischDEexp(twr, f, g, x, theta, driv, limint, extint) ==
(u := monomRDE(f, g, driv)) case "failed" => "failed"
(v := gpolDEexp(twr, u.a, RF2GP(u.b), RF2GP(u.c), x, theta, driv,
   limint, extint)) case "failed" => "failed"
convert(v::GP)*RF / u.t::RF

polyDElog(twr, aa, bb, cc, x, t, driv, limint, extint) ==
zero? cc => 0
t' := differentiate(t::F, x)
zero? bb =>
(u := cc exquo aa) case "failed" => "failed"
primintfldpoly(u::UP, z1 +-> extint(z1, t'), t')
n := degree(cc)::Z - (db := degree(bb)::Z)
if ((da := degree(aa)::Z = db) and (da > 0) then
  lk0 := tower(f0 :=
    -(leadingCoefficient bb) / (leadingCoefficient aa))
lk1 := logdiff(twr, lk0)
  (if0 := limint(lk0, [first argument u for u in lk1]))
    case "failed" => error "Risch's theorem violated"
  alph := validExponential(lk0, RRF2F(if0::RRF), x)) case F =>
    return
    (ans := polyDElog(twr, alph::F * aa,
      differentiate(alph::F, x) * aa + alph::F * bb,
      cc, x, t, driv, limint, extint)) case "failed" => "failed"
    alph::F * ans::UP
  if (da > db + 1) then n := max(0, degree(cc)::Z - da + 1)
  if (da = db + 1) then
    i := limint(- (leadingCoefficient bb) / (leadingCoefficient aa),
      [first argument t])
  if not(i case "failed") then
    r :=
      null(i.limitedlogs) => 0$F
      i.limitedlogs.first.coeff
    if (nn := retractIfCan(r)@Union(Z, "failed")) case Z then
      n := max(nn::Z, n)
    (v := polyRDE(aa, bb, cc, n, driv)) case ans =>
      v.ans.nosol => "failed"
      v.ans.ans
    w := v.eq
    zero?(w.b) =>
      degree(w.c) > w.m => "failed"
      (u := primintfldpoly(w.c, z1 +-> extint(1, t'), t'))
      case "failed" => "failed"
degree(u::UP) > w.m => "failed"
w.alpha * u::UP + w.beta
(u := logdegrad(twr, retract(w.b), w.c, w.m, x, t, limint, extint))
case "failed" => "failed"
w.alpha * u::UP + w.beta

expintfdpoly(u::GP,
(z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
lb := boundAt0(twr, - coefficient(b, 0) / coefficient(a, 0),
   nb := order b, nc := order c, x, t, limint)
tm := monomial(1, (m := max(0, max(-nb, lb - nc))))::N$UP
(v := polyDEexp(twr,a * tm,lb * differentiate(first argument t, x)
   * a * tm + retract(b * tm::GP)$UP,
   retract(c * monomial(1, m - lb))$UP,
   x, t, driv, limint, extint)) case "failed" => "failed"
v::UP::GP * monomial(1, lb)

gpolDEexp(twr, a, b, c, x, t, driv, limint, extint) ==
zero? c => 0
zero? b =>
   (u := c exquo (a::GP)) case "failed" => "failed"
expintfdpoly(u::GP,
   (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
1b := boundAt0(twr, - coefficient(b, 0) / coefficient(a, 0),
   nb := order b, nc := order c, x, t, limint)
tm := monomial(1, (m := max(0, max(-nb, lb - nc))))::N$UP
(v := polyDEexp(twr,a * tm,lb * differentiate(first argument t, x)
   * a * tm + retract(b * tm::GP)$UP,
   retract(c * monomial(1, m - lb))$UP,
   x, t, driv, limint, extint)) case "failed" => "failed"
v::UP::GP * monomial(1, lb)

expintfdpoly(u::GP, rischdiffeq)) case "failed" => "failed"
retractIfCan(u::GP)$Union(UP, "failed")

boundInf(twr, f0, da, db, dc, x, t, limitedint) ==
da < db => dc - db
da > db => max(0, dc - da)
CHAPTER 6. CHAPTER E

\[ l_1 := \text{logdiff}(t \text{wr}, l_0 := \text{tower } f_0) \]

(if0 := \text{limitedint}(f_0, \text{[first argument } u \text{ for } u \text{ in } l_1))
  \text{case } "failed" \Rightarrow \text{error } "\text{Risch’s theorem violated}" \\
  (alpha := \text{validExponential}(\text{concat}(t, l_0), \text{RRF2F}(\text{if0::RRF}), x))

\begin{align*}
\text{case } F & \Rightarrow \\
  a1 := \text{separate(\text{univariate}(alpha::F, t))\$GP} & \\
  \text{zero?(a1.fracPart) and monomial?(a1.polyPart) } \Rightarrow \\
  \text{max}(0, \text{max}(\text{degree}(a1.polyPart), dc - db)) \\
\end{align*}

\[ dc - db \]

boundAt0(t \text{wr}, f_0, nb, nc, x, t, \text{limitedint}) == 
\begin{align*}
\text{nb} & \neq 0 \Rightarrow \text{min}(0, nc - \text{min}(0, nb)) \\
\end{align*}

\[ l_1 := \text{logdiff}(t \text{wr}, l_0 := \text{tower } f_0) \]

(if0 := \text{limitedint}(f_0, \text{[first argument } u \text{ for } u \text{ in } l_1))
  \text{case } "failed" \Rightarrow \text{error } "\text{Risch’s theorem violated}" \\
  (alpha := \text{validExponential}(\text{concat}(t, l_0), \text{RRF2F}(\text{if0::RRF}), x))

\begin{align*}
\text{case } F & \Rightarrow \\
  u1 := \text{primintfldpoly}(\text{inv}(alpha::F) * c, z1+->\text{extint}(z1, t'), t') & \\
  \text{case } "failed" \Rightarrow "\text{failed}"
\end{align*}

\[ \text{degree}(u1::UP)::Z > n \Rightarrow "\text{failed}" \]

\[ \alpha::F = u1::UP \]

logdeg(c, - if0.mainpart - \\
  +/[v.coeff * log(v.logand) for v in if0.limitedlogs], \\
  n, x, t', \text{limitedint}, \text{extint})

-- case a = 1, degree(b) = 0, and (exp integrate b) is not in F
-- this implies no cancellation at infinity

\begin{align*}
\text{logdeg}(c, f, n, x, t', \text{limitedint}, \text{extint}) & := \\
\text{answr:UP} & := 0 \\
\text{repeat} \\
\text{zero?(c) } \Rightarrow \text{return answr} \\
(n < 0) \text{ or } (m := \text{degree } c)::Z > n) \Rightarrow \text{return } "\text{failed}"
\end{align*}

\[ u := \text{rischDE}(1, f, \text{leadingCoefficient } c, x, \text{limitedint}, \text{extint}) \]

\[ u.sol? \Rightarrow \text{return } "\text{failed}"
\]

\[ \text{zero?(m) } \Rightarrow \text{return}(\text{answr } + u.ans::UP) \]
\[ n := m \cdot Z - 1 \]
\[ c := (\text{reductum } c) - \text{monomial}(m \cdot Z \ast t' \ast u.\text{ans}, (m - 1) \cdot N) \]
\[ \text{answr} := \text{answr} + \text{monomial}(u.\text{ans}, m) \]

-- case \( a = 1 \), \( \deg(B) = 0 \), \( B \neq 0 \)
-- cancellation at infinity is possible

\[
\text{expdegrad}(\text{twr}, b, c, n, x, t, \text{limint}, \text{extint}) ==
\]
\[
\begin{align*}
\text{lk1} & := \text{logdiff}(\text{twr}, \text{lk0} := \text{tower}(f0 := -b)) \\
(\text{if0} & := \text{limint}(f0, [\text{first argument } u \text{ for } u \text{ in } \text{lk1}]))
\end{align*}
\]
\[
\begin{cases}
\text{case "failed"} & \Rightarrow \text{error "Risch's theorem violated"} \\
\text{intf0} & := - \text{if0.mainpart} - \\
\quad +/[v.\text{coeff} \ast \log(v.\text{logand}) \text{ for } v \text{ in } \text{if0.limitedlogs}] \\
(\alpha & := \text{validExponential}(\text{concat}(t, 1k0), \text{RRF2F}(\text{if0}::\text{RRF}), x))
\end{cases}
\]
\[
\begin{cases}
\text{case } F & => \\
\quad \text{al} & := \text{separate(\text{univariate}(\text{alpha}::F, t)))}\$GP \\
\quad \text{zero?}(\text{al.fracPart}) \text{ and } \text{monomial?}(\text{al.polyPart}) \text{ and }
\quad (\text{degree}(\text{al.polyPart}) >= 0) \Rightarrow \\
\quad (\text{ui} & := \text{expintfldpoly}(c::GP \ast \text{recip}(\text{al.polyPart})::\text{GP}, \\
\quad (z1, z2) & \mapsto \text{rischDE}(z1, \text{first argument } t, z2, x, \text{limint}, \text{extint})))
\end{cases}
\]
\[
\begin{cases}
\text{case "failed"} & \Rightarrow "\text{failed}" \\
\quad \text{degree}(\text{ui}::\text{GP}) > n & \Rightarrow "\text{failed}"
\end{cases}
\]
\[
\text{retractIfCan(\text{al.polyPart} * \text{ui}::\text{GP})@Union(UP, "\text{failed}")}
\]
\[
\text{expdeg}(c, \text{intf0}, n, x, \text{first argument } t, \text{limint}, \text{extint})
\]

-- case \( a = 1 \), \( \deg(B) = 0 \), and \( (\exp \text{integrate } b) \) is not a monomial
-- this implies no cancellation at infinity

\[
\text{expdeg}(c, f, n, x, \eta, \text{limitedint}, \text{extint}) ==
\]
\[
\begin{align*}
\text{answr} & := UP := 0 \\
\text{repeat}
\end{align*}
\]
\[
\begin{align*}
\text{zero? } c & \Rightarrow \text{return } \text{answr} \\
(n < 0) \text{ or } (m := \text{degree } c)::Z > n & \Rightarrow \text{return "failed"
\quad u := \text{rischDE}(1, f + m \ast \eta, \text{leadingCoefficient } c, x, \text{limitedint}, \text{extint})
\quad \text{"u.sol?} & \Rightarrow \text{return "failed"}
\quad \text{zero? } m & \Rightarrow \text{return(\text{answr} + u.\text{ans}::UP) \\
\quad n & := m::Z - 1 \\
\quad c & := \text{reductum } c \\
\quad \text{answr} & := \text{answr} + \text{monomial}(u.\text{ans}, m) \]
\end{align*}
\]

\[
\text{RRF2F } \text{rrf} ==
\]
\[
\begin{align*}
\text{rrf.mainpart} & +/[v.\text{coeff}\ast\log(v.\text{logand}) \text{ for } v \text{ in } \text{rrf.limitedlogs}]
\end{align*}
\]

\[
\text{—— RDEEF.dotabb ——}
\]

"RDEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDEEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
package RDEEFS ElementaryRischDESystem

--- ElementaryRischDESystem.input ---

)set break resume
)sys rm -f ElementaryRischDESystem.output
)spool ElementaryRischDESystem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryRischDESystem
--E 1

)spool
)lisp (bye)

---

--- ElementaryRischDESystem.help ---

====================================================================
ElementaryRischDESystem examples
====================================================================

Risch differential equation, elementary case.

See Also:
  o )show ElementaryRischDESystem

---
ElementaryRischDESystem (RDEEFS)
++ lim is a limited integration function,
++ ext is an extended integration function.

Implementation ==>
import IntegrationTools(R, F)
import ElementaryRischDE(R, F)
import TranscendentalRischDESystem(F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

-- sm1 := sqrt(-1::F)
-- ks1 := retract(sm1)@K
-- gcoeffs : P -> ULF
-- gets1coeffs: F -> ULF
-- cheat : (Z, F, F, F, SE, (F, List F) -> U, (F, F) -> UEX) -> ULF
basecase : (F, F, F, K) -> ULF

basecase(nfp, g1, g2, k) ==
(baseRDEsys(univariate(nfp, k), univariate(g1, k),
univariate(g2, k))) case "failed" => "failed"
l := ans::List(RF)
[multivariate(first l, k), multivariate(second l, k)]

-- returns [x,y] s.t. f = x + y %i
-- f can be of the form (a + b %i) / (c + d %i)
-- gets1coeffs f ==
-- (lnum := gcoeffs(numer f)) case "failed" => "failed"
-- (lden := gcoeffs(denom f)) case "failed" => "failed"
-- a := first(lnum::List F)
-- b := second(lnum::List F)
-- c := first(lden::List F)
-- zero?(d := second(lden::List F)) => [a/c, b/c]
-- cd := c * c + d * d
-- [(a * c + b * d) / cd, (b * c - a * d) / cd]

-- gcoeffs p ==
-- degree(q := univariate(p, ks1)) > 1 => "failed"
-- [coefficient(q, 0)::F, coefficient(q, 1)::F]

-- cheat(n, f, g1, g2, x, limint, extint) ==
-- (u := rischDE(n, sm1 * f, g1 + sm1 * g2, x, limint, extint))
-- case "failed" => "failed"
-- (l := gets1coeffs(u::F)) case "failed" =>
-- error "rischDEsys: expect linear result in sqrt(-1)"
-- l::List F

-- solve (y1',y2') + ((0, -n f'), (n f', 0)) (y1,y2) = (g1, g2)
rischDEsys(n, f, g1, g2, x, limint, extint) ==
zero? g1 and zero? g2 => [0, 0]
zero?(nfp := n * differentiate(f, x)) => 
  ((u1 := limint(g1, empty())) case "failed") or 
  ((u2 := limint(g1, empty())) case "failed") => "failed"
  [u1.mainpart, u2.mainpart]
freeOf?(y1 := g2 / nfp, x) and freeOf?(y2 := -g1 / nfp, x) => [y1, y2]
v1 := varselect(union(kernels nfp, union(kernels g1, kernels g2)), x)
symbolIfCan(k := kmax v1) case SE => basecase(nfp, g1, g2, k)
-- cheat(n, f, g1, g2, x, limint, extint)
error "rischDEsys: can only handle rational functions for now"

---

package ELFUTS EllipticFunctionsUnivariateTaylorSeries

--- EllipticFunctionsUnivariateTaylorSeries.input ---

)set break resume
)sys rm -f EllipticFunctionsUnivariateTaylorSeries.output
)spool EllipticFunctionsUnivariateTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EllipticFunctionsUnivariateTaylorSeries
--E 1

)spool
)lisp (bye)

---

package ELFUTS EllipticFunctionsUnivariateTaylorSeries.help
The elliptic functions \( sn \), \( sc \) and \( dn \) are expanded as Taylor series.

See Also:
- \( )\)show EllipticFunctionsUnivariateTaylorSeries

---

**EllipticFunctionsUnivariateTaylorSeries (ELFUTS)**

Exports:
- \( cn \)
- \( dn \)
- \( sn \)
- \( sn \) and \( dn \)

---

\( \)abbrev package ELFUTS EllipticFunctionsUnivariateTaylorSeries
++ Author: Bill Burge, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 17 February 1992
++ Description:
++ The elliptic functions \( sn \), \( sc \) and \( dn \) are expanded as Taylor series.

EllipticFunctionsUnivariateTaylorSeries(Coef, UTS):
Exports == Implementation where
\begin{align*}
\text{Coef} & : \text{Field} \\
\text{UTS} & : \text{UnivariateTaylorSeriesCategory Coef}
\end{align*}
\begin{align*}
L & => \text{List} \\
I & => \text{Integer} \\
\text{RN} & => \text{Fraction Integer}
\end{align*}
ST ==> Stream Coef
STT ==> StreamTaylorSeriesOperations Coef
YS ==> Y$ParadoxicalCombinatorsForStreams(Coef)
Exports ==> with
   sn : (UTS,Coef) -> UTS
   \spad{sn(x,k)} expands the elliptic function sn as a Taylor
   series.
   cn : (UTS,Coef) -> UTS
   \spad{cn(x,k)} expands the elliptic function cn as a Taylor
   series.
   dn : (UTS,Coef) -> UTS
   \spad{dn(x,k)} expands the elliptic function dn as a Taylor
   series.
   sncndn: (ST,Coef) -> L ST
   \spad{sncndn(s,c)} is used internally.
Implementation ==> add
import StreamTaylorSeriesOperations Coef
UPS ==> StreamTaylorSeriesOperations Coef
integrate ==> lazyIntegrate
sncndnre:(Coef,L ST,ST,Coef) -> L ST
sncndnre(k,scd,dx,sign) ==
   [integrate(0,   scd.2*$UPS scd.3*$UPS dx), _
    integrate(1, sign*scd.1*$UPS scd.3*$UPS dx), _
    integrate(1,sign*k**2*$UPS scd.1*$UPS scd.2*$UPS dx)]

sncndn(z,k) ==
   empty? z => [0 :: ST,1 :: ST,1::ST]
   frst z = 0 => YS(x +-> sncndnre(k,x,deriv z,-1),3)
   error "ELFUTS:sncndn: constant coefficient should be 0"
   sn(x,k) == series sncndn.(coefficients x,k).1
   cn(x,k) == series sncndn.(coefficients x,k).2
   dn(x,k) == series sncndn.(coefficients x,k).3

---

— ELFUTS.dotabb —

"ELFUTS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ELFUTS"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"ELFUTS" -> "UTSCAT"
package EQ2 EquationFunctions2

--- EquationFunctions2.input ---

)set break resume
/sys rm -f EquationFunctions2.output
/spool EquationFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show EquationFunctions2
--E 1

)spool
)lisp (bye)

---

--- EquationFunctions2.help ---

====================================================================
EquationFunctions2 examples
====================================================================

This package provides operations for mapping the sides of equations.

See Also:
  o )show EquationFunctions2

---
EquationFunctions2 (EQ2)

Exports:
  map

--- package EQ2 EquationFunctions2 ---

)abbrev package EQ2 EquationFunctions2
++ Date Last Updated: June 3, 1991
++ Description:
++ This package provides operations for mapping the sides of equations.

EquationFunctions2(S: Type, R: Type): with
  map: (S -> R, Equation S) -> Equation R
  ++ map(f, eq) returns an equation where f is applied to the sides of eq
  == add
  map(fn, eqn) == equation(fn lhs eqn, fn rhs eqn)

--- EQ2.dotabb ---

"EQ2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EQ2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"EQ2" -> "TYPE"

---

package ERROR ErrorFunctions

--- ErrorFunctions.input ---
ErrorFunctions implements error functions callable from the system interpreter. Typically, these functions would be called in user functions. The simple forms of the functions take one argument which is either a string (an error message) or a list of strings which all together make up a message. The list can contain formatting codes (see below). The more sophisticated versions takes two arguments where the first argument is the name of the function from which the error was invoked and the second argument is either a string or a list of strings, as above. When you use the one argument version in an interpreter function, the system will automatically insert the name of the function as the new first argument. Thus in the user interpreter function

\[
\text{f x == if x < 0 then error "negative argument" else x}
\]

the call to error will actually be of the form

\[
\text{error("f","negative argument")}
\]

because the interpreter will have created a new first argument.

Formatting codes: error messages may contain the following formatting codes (they should either start or end a string or else have blanks around them):

- %l: start a new line
- %b: start printing in a bold font (where available)
%d  stop printing in a bold font (where available)
%ceon  start centering message lines
%ceoff  stop centering message lines
%rjon  start displaying lines "ragged left"
%rjoff  stop displaying lines "ragged left"
%i  indent following lines 3 additional spaces
%u  unindent following lines 3 additional spaces
%xN  insert N blanks (eg, %x10 inserts 10 blanks)

Examples:
1. error "Whoops, you made a %l %ceon %b big %d %ceoff %l mistake!"
2. error ["Whoops, you made a",%l %ceon %b","big",
   "%d %ceoff %l","mistake!"]

See Also:
  o )show ErrorFunctions

---

ErrorFunctions (ERROR)

Exports:
error

--- package ERROR ErrorFunctions ---

)abbrev package ERROR ErrorFunctions
++ Author: Robert S. Sutor
++ Date Created: 29 May 1990
++ Date Last Updated: 29 May 1990
++ Description:
++ ErrorFunctions implements error functions callable from the system
++ interpreter. Typically, these functions would be called in user
functions. The simple forms of the functions take one argument
which is either a string (an error message) or a list of strings
which all together make up a message. The list can contain
formatting codes (see below). The more sophisticated versions takes
two arguments where the first argument is the name of the function
from which the error was invoked and the second argument is either a
string or a list of strings, as above. When you use the one
argument version in an interpreter function, the system will
automatically insert the name of the function as the new first
argument. Thus in the user interpreter function:

```
\texttt{f \langle x \rangle == if x < 0 \text{ then error \"negative argument\" else x\rangle}
```
the call to error will actually be of the form:

```
\texttt{error("f","negative argument")}
```
because the interpreter will have created a new first argument.

++ Formatting codes: error messages may contain the following
++ formatting codes (they should either start or end a string or
++ else have blanks around them):
++ \texttt{\%l} \tab{6} start a new line
++ \texttt{\%b} \tab{6} start printing in a bold font (where available)
++ \texttt{\%d} \tab{6} stop printing in a bold font (where available)
++ \texttt{\%ceon} \tab{3} start centering message lines
++ \texttt{\%ceoff} \tab{2} stop centering message lines
++ \texttt{\%rjon} \tab{3} start displaying lines "ragged left"
++ \texttt{\%rjoff} \tab{2} stop displaying lines "ragged left"
++ \texttt{\%i} \tab{6} indent following lines 3 additional spaces
++ \texttt{\%u} \tab{6} unindent following lines 3 additional spaces
++ \texttt{\%xN} \tab{5} insert N blanks (eg, \texttt{\%x10} inserts 10 blanks)
++
++ Examples:
++ 1.\texttt{error \"Whoops, you made a %l %ceon %b big %d %ceoff %l mistake!\"}
++ 2.\texttt{error [\"Whoops, you made a\",%l %ceon %b","big","%d %ceoff %l","mistake!\"]}

ErrorFunctions() : Exports == Implementation where
Exports ==

```
\texttt{error: String -> Exit}
\texttt{error\langle msg \rangle \textit{ displays error message } msg \textit{ and terminates.}}
\texttt{error: List String -> Exit}
\texttt{error\langle lmsg \rangle \textit{ displays error message } lmsg \textit{ and terminates.}}
\texttt{error: (String,String) -> Exit}
\texttt{error\langle (nam,msg) \rangle \textit{ displays error message } msg \textit{ preceded by a}}
\texttt{message containing the name } nam \textit{ of the function in which}}
\texttt{the error is contained.}}
\texttt{error: (String,List String) -> Exit}
\texttt{error\langle (nam,lmsg) \rangle \textit{ displays error messages } lmsg \textit{ preceded by a}}
\texttt{message containing the name } nam \textit{ of the function in which}}
\texttt{the error is contained.}}
```
Implementation ==

add
prefix1 : String := "Error signalled from user code: %l "
prefix2 : String := "Error signalled from user code in function %b "

doit(s : String) : Exit ==
throwPatternMsg(s,nil$(List String))$Lisp
   -- there are no objects of type Exit, so we'll fake one,
   -- knowing we will never get to this step anyway.
   "exit" pretend Exit

error(s : String) : Exit ==
doit concat [prefix1,s]

error(l : List String) : Exit ==
s : String := prefix1
   for x in l repeat s := concat [s," ",x]
doit s

error(fn : String, s : String) : Exit ==
doit concat [prefix2,fn,: %d %l ",s]

error(fn : String, l : List String) : Exit ==
s : String := concat [prefix2,fn,: %d %l"
   for x in l repeat s := concat [s," ",x]
doit s

———

— ERROR.dotabb —

"ERROR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ERROR"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"ERROR" -> "STRING"

———

package GBEUCLID EuclideanGroebnerBasisPackage

— EuclideanGroebnerBasisPackage.input —

)set break resume
)sys rm -f EuclideanGroebnerBasisPackage.output
)spool EuclideanGroebnerBasisPackage.output
)set message test on
)set message auto off
\( \text{clear all} \)

\( \text{--S 1 of 24} \)
\( a1: \text{DMP}([y,x], \text{INT}):= (9x^2 + 5x - 3) + y(3x^2 + 2x + 1) \)
\( \text{--R} \)
\( \text{--R 2 2} \)
\( \text{--R (1) 3y x + 2y x + y + 9x + 5x - 3} \)
\( \text{--R Type: DistributedMultivariatePolynomial([y,x],Integer)} \)
\( \text{--E 1} \)

\( \text{--S 2 of 24} \)
\( a2: \text{DMP}([y,x], \text{INT}):= (6x^3 - 2x^2 - 3x +3) + y(2x^3 - x - 1) \)
\( \text{--R} \)
\( \text{--R 3 3 2} \)
\( \text{--R (2) 2y x - y x - y + 6x - 2x - 3x + 3} \)
\( \text{--R Type: DistributedMultivariatePolynomial([y,x],Integer)} \)
\( \text{--E 2} \)

\( \text{--S 3 of 24} \)
\( a3: \text{DMP}([y,x], \text{INT}):= (3x^3 + 2x^2) + y(x^3 + x^2) \)
\( \text{--R} \)
\( \text{--R 3 2 3 2} \)
\( \text{--R (3) y x + y x + 3x + 2x} \)
\( \text{--R Type: DistributedMultivariatePolynomial([y,x],Integer)} \)
\( \text{--E 3} \)

\( \text{--S 4 of 24} \)
\( an:= [a1,a2,a3] \)
\( \text{--R} \)
\( \text{--R (4) 2 2 3 3 2} \)
\( \text{--R [3y x + 2y x + y + 9x + 5x - 3, 2y x - y x - y + 6x - 2x - 3x + 3,} \)
\( \text{--R 3 2 3 2} \)
\( \text{--R y x + y x + 3x + 2x]} \)
\( \text{--R Type: List(DistributedMultivariatePolynomial([y,x],Integer))} \)
\( \text{--E 4} \)

\( \text{--S 5 of 24} \)
\( \text{euclideanGroebner(an)} \)
\( \text{--R} \)
\( \text{--R 2 3 2} \)
\( \text{--R (5) [y x - y + x + 3,2y + 2x - 3x - 6,2x - 5x - 5]} \)
\( \text{--R Type: List(DistributedMultivariatePolynomial([y,x],Integer))} \)
\( \text{--E 5} \)

\( \text{--S 6 of 24} \)
\( \text{euclideanGroebner(an,"redcrit")} \)
\( \text{--R} \)
\( \text{--R} \)
\( \text{--R reduced Critpair - Polynom :} \)
\( \text{--R} \)
```plaintext
--R 2
--R - 2y x - y x - y - 6x - 3x + 3
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R y x - y + x + 3
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 4y + 4x - 6x - 12
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 4x + 10x + 10x
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 2y + 2x - 3x - 6
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
```
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 2x + 5x + 5x
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R 2 3 2
--R (6) [y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x]
--R Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 6

--S 7 of 24
euclideanGroebner(an,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R 3 3 2 2
--R [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R 2 2
--R [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x ,tc= 4,rc= y x ,trc= 4,tH= 1,tD= 3]]
--R
--R 2
--R [[ci= y x ,tci= 6,cj= y x ,tcj= 4,c= y x ,tc= 5,rc= y,trc= 4,tH= 2,tD= 3]]
--R
--R 3
--R [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R 2
--R [[ci= y x ,tci= 6,cj= y x ,tcj= 4,c= y x ,tc= 5,rc= y,trc= 4,tH= 3,tD= 4]]
--R
--R 2
--R [[ci= y,tci= 4,cj= y,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
--R
--R 3
--R [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R 3 3
--R [[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 2]]
--R
--R 3
--R [[ci= y x ,tci= 4,cj= y x ,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R 3 2
--R [[ci= y,tcj= 4,c= y x ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R There are
--R
--R 3
---R Groebner Basis Polynomials.
---R
---R THE GROEBNER BASIS over EUCLIDEAN DOMAIN
---R
---R (7) \([y^2 x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x]\)
---R Type: List(DistributedMultivariatePolynomial([y, x], Integer))
---E 7

---S 8 of 24

euclideanGroebner(an, "info", "redcrit")
---R
---R reduced Critpair - Polynom :
---R
---R
---R 2 2
---R - 2y x - y x - y - 6x - 3x + 3
---R
---R
---R
---R you choose option -info-
---R abbrev. for the following information strings are
---R ci => Leading monomial for critpair calculation
---R tci => Number of terms of polynomial i
---R cj => Leading monomial for critpair calculation
---R tcj => Number of terms of polynomial j
---R c => Leading monomial of critpair polynomial
---R tc => Number of terms of critpair polynomial
---R rc => Leading monomial of redcritpair polynomial
---R trc => Number of terms of redcritpair polynomial
---R tF => Number of polynomials in reduction list F
---R tD => Number of critpairs still to do
---R
---R
---R
---R
---R
---R
---R
---R
---R
---R
---R

---R 3 3 2 2
---R \([\text{ci=} y x ,\text{tci=} 7, \text{cj=} y x ,\text{tcj=} 4, c= y x ,tc= 6, rc= y x ,trc= 6, tF= 3, tD= 3]\)
---R
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R y x - y + x + 3
---R
--R
--R
--R 2       2
--R [[ci= y x, tci= 6, cj= y x, tcj= 6, c= y x, tc= 4, rc= y x, trc= 4, tH= 1, tD= 3]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 4y + 4x - 6x - 12
--R
--R
--R
--R 2
--R [[ci= y x, tci= 6, cj= y x, tcj= 4, c= y x, tc= 5, rc= y, trc= 4, tH= 2, tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3
--R - 4x + 10x + 10x
--R
--R
--R
--R
--R 3
--R [[ci= y x, tci= 4, cj= y, tcj= 4, c= y, tc= 5, rc= x, trc= 3, tH= 3, tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 2y + 2x - 3x - 6
--R
--R
--R
--R 2
--R [[ci= y x, tci= 6, cj= y x, tcj= 4, c= y x, tc= 5, rc= y, trc= 4, tH= 3, tD= 4]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
reduced Critpair - Polynom:

\[3 \quad 2\]
\[-2x + 5x + 5x\]

reduced Critpair - Polynom:

\[3\]
\[-y, tci= 4, cj= y, tcj= 4, c= 0, tc= 0, trc= 0, tH= 3, tD= 3]\]

reduced Critpair - Polynom:

\[0\]

reduced Critpair - Polynom:

\[3 \quad 3\]
\[-w, tci= 3,cj= w, tcj= 3, c= 0, tc= 0, trc= 0, tH= 3, tD= 2]\]

reduced Critpair - Polynom:

\[0\]

reduced Critpair - Polynom:

\[3 \quad 2\]
\[-y, tci= 4, cj= y, tcj= 4, c= y, tc= 3, trc= 0, tH= 3, tD= 1]\]

reduced Critpair - Polynom:

\[0\]
There are 3 Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(8) \[y \cdot x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x\]

Type: List(DistributedMultivariatePolynomial([y,x],Integer))

b1: HDMP([y,x],INT):= (9*x**2 + 5*x - 3) + y*(3*x**2 + 2*x + 1)

(9) \[3y \cdot x + 2y \cdot x + 9x + y + 5x - 3\]

Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)

b2: HDMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x + 3) + y*(2*x**3 - x - 1)

(10) \[2y \cdot x + 6x - y \cdot x - 2x - y - 3x + 3\]

Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)

b3: HDMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)

(11) \[y \cdot x + y \cdot x + 3x + 2x\]

Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)

bn:=[b1,b2,b3]

(12) \[3y \cdot x + 2y \cdot x + 9x + y + 5x - 3, 2y \cdot x + 6x - y \cdot x - 2x - y - 3x + 3, 3y \cdot x + 2y \cdot x + 9x + y + 5x - 3, 2y \cdot x + 6x - y \cdot x - 2x - y - 3x + 3,\]
--R  y x + y x + 3x + 2x ]
--R  Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 12

--S 13 of 24
euclideanGroebner(bn)
--R
--R  2
--R  (13) [2y - 5y - 8x - 3,y x - y + x + 3,2x + 2y - 3x - 6]
--R  Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 13

--S 14 of 24
euclideanGroebner(bn,"redcrit")
--R
--R
--R  reduced Critpair - Polynom :
--R
--R
--R  2 2
--R  - 2y x - y x - 6x - y - 3x + 3
--R
--R
--R
--R  reduced Critpair - Polynom :
--R
--R
--R
--R  y x - y + x + 3
--R
--R
--R
--R  reduced Critpair - Polynom :
--R
--R
--R
--R  y x - y + x + 3
--R
--R
--R
--R  reduced Critpair - Polynom :
--R
--R
--R  2
--R  4x + 4y - 6x - 12
--R
--R
--R
--R
--R  reduced Critpair - Polynom :
--R
--R
--R  2
--R  2x + 2y - 3x - 6
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R 2 2
--R (14) [2y - 5y - 8x - 3, y x - y + x + 3, 2x + 2y - 3x - 6]
--R Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 14

--S 15 of 24
euclideanGroebner(bn,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
There are 3 Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN
euclideanGroebner(bn,"info","redcrit")

reduced Critpair - Polynom:

\[ 2 \cdot 2 \]
\[ - 2y x - y x - 6x - y - 3x + 3 \]

you choose option -info-

<table>
<thead>
<tr>
<th>Abbrev. for the following information strings are</th>
</tr>
</thead>
<tbody>
<tr>
<td>ci =&gt; Leading monomial for critpair calculation</td>
</tr>
<tr>
<td>tci =&gt; Number of terms of polynomial i</td>
</tr>
<tr>
<td>cj =&gt; Leading monomial for critpair calculation</td>
</tr>
<tr>
<td>tcj =&gt; Number of terms of polynomial j</td>
</tr>
<tr>
<td>c =&gt; Leading monomial of critpair polynomial</td>
</tr>
<tr>
<td>tc =&gt; Number of terms of critpair polynomial</td>
</tr>
<tr>
<td>rc =&gt; Leading monomial of redcritpair polynomial</td>
</tr>
<tr>
<td>trc =&gt; Number of terms of redcritpair polynomial</td>
</tr>
<tr>
<td>tF =&gt; Number of polynomials in reduction list F</td>
</tr>
<tr>
<td>tD =&gt; Number of critpairs still to do</td>
</tr>
</tbody>
</table>

\[ 3 \cdot 3 \cdot 2 \cdot 2 \]
\[ \{[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]\} \]

reduced Critpair - Polynom:

\[ y x - y + x + 3 \]

\[ 2 \cdot 2 \]
\[ \{[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]\} \]
reduced Critpair - Polynom:
2
4x + 4y - 6x - 12

2
[[ci = \ y \ x , tci = 6, cj = \ y \ x, tcj = 4, c = \ y \ x, tc = 5, rc = \ x , trc = 4, tH = 2, tD = 3]]

reduced Critpair - Polynom:
2
2x + 2y - 3x - 6

2
[[ci = \ y \ x , tci = 6, cj = \ y \ x, tcj = 4, c = \ y \ x, tc = 5, rc = \ x , trc = 4, tH = 2, tD = 3]]

reduced Critpair - Polynom:
0

2
[[ci = \ x \ , tci = 4, cj = \ x , tcj = 4, c = 0, tc = 0, rc = 0, trc = 0, tH = 2, tD = 2]]

reduced Critpair - Polynom:
2
- 2y + 5y + 8x + 3

2
2
[[ci = \ y \ x, tci = 4, cj = \ x , tcj = 4, c = \ y , tc = 5, rc = \ y , trc = 4, tH = 3, tD = 2]]
reduced Critpair - Polynom :
0

2 2
[[ci= y x,tci= 4,cj= y ,tcj= 4,c= y ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 1]]

reduced Critpair - Polynom :
0

3
[[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 0]]

There are
3
Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(16) [2y - 5y - 8x - 3,y x - y + x + 3,2x + 2y - 3x - 6]
Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))

(17) 3y x + 2y x + y + 9x + 5x - 3
Type: GeneralDistributedMultivariatePolynomial([y,x],Integer,...
c3: GMP([y,x], INT, DIRPROD(2, NNI)) := (3*x^3 + 2*x^2) + y*(x^3 + x^2)

S 19 of 24

cn:=[c1,c2,c3]

S 20 of 24

euclideanGroebner(cn)

S 21 of 24

S 22 of 24

euclideanGroebner(cn,"redcrit")
PACKAGE GBEUCLID EUCLIDEANGROEBNERBASIS PACKAGE

--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 4y + 4x - 6x - 12
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 4x + 10x + 10x
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 2y + 2x - 3x - 6
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 2x + 5x + 5x
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
reduced Critpair - Polynom :

reduced Critpair - Polynom :

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(22) \[ y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x \]

Type: List(GeneralDistributedMultivariatePolynomial([y, x], Integer,...

you choose option -info-

abbrev. for the following information strings are:

- ci => Leading monomial for critpair calculation
- tci => Number of terms of polynomial i
- cj => Leading monomial for critpair calculation
- tcj => Number of terms of polynomial j
- c => Leading monomial of critpair polynomial
- tc => Number of terms of critpair polynomial
- rc => Leading monomial of redcritpair polynomial
- trc => Number of terms of redcritpair polynomial
- tF => Number of polynomials in reduction list F
- tD => Number of critpairs still to do

[[ci= y x , tci= 7, cj= y x , tcj= 4, c= y x , tc= 6, rc= y x , trc= 6, tF= 3, tD= 3]]
There are 3 Groebner Basis Polynomials.

The Groebner Basis over Euclidean Domain

(23) \[y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x\]

Type: List(GeneralDistributedMultivariatePolynomial([y, x], Integer, ...)

S 24 of 24
euclideanGroebner(cn,"info","redcrit")
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2 2
--R - 2y x - y x - y - 6x - 3x + 3
--R
--R
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R
--R
--R 3 3 2 2
--R [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R y x - y + x + 3
--R
--R
--R
--R 2 2
--R [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x ,tc= 4,rc= y x ,trc= 4,tH= 1,tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R 2
--R 4y + 4x - 6x - 12
2

[[ci = y x, tci = 6, cj = y x, tcj = 4, c = y x, tc = 5, rc = y, trc = 4, tH = 2, tD = 3]]

reduced Critpair - Polynom :

3

[[ci = y x, tci = 4, cj = y, tcj = 4, c = y, tc = 5, rc = x, trc = 3, tH = 3, tD = 3]]

reduced Critpair - Polynom :

2

2y + 2x - 3x - 6

reduced Critpair - Polynom :

2

[[ci = y x, tci = 6, cj = y x, tcj = 4, c = y x, tc = 5, rc = y, trc = 4, tH = 3, tD = 4]]

reduced Critpair - Polynom :

0

[[ci = y, tci = 4, cj = y, tcj = 4, c = 0, tc = 0, rc = 0, trc = 0, tH = 3, tD = 3]]

reduced Critpair - Polynom :

3

[[ci = y, tci = 4, cj = y, tcj = 4, c = 0, tc = 0, rc = 0, trc = 0, tH = 3, tD = 3]]

reduced Critpair - Polynom :

3

- 2x + 5x + 5x
reduced Critpair - Polynom :

There are 3 Groebner Basis Polynomials.
EuclideanGroebnerBasisPackage computes groebner bases for polynomial ideals over euclidean domains. The basic computation provides a distinguished set of generators for these ideals. This basis allows an easy test for membership: the operation euclideanNormalForm returns zero on ideal members. The string "info" and "redcrit" can be given as additional args to provide incremental information during the computation.

If "info" is given, a computational summary is given for each s-polynomial. If "redcrit" is given, the reduced critical pairs are printed.

The term ordering is determined by the polynomial type used.
Suggested types include
* DistributedMultivariatePolynomial
* HomogeneousDistributedMultivariatePolynomial
* GeneralDistributedMultivariatePolynomial

Example to call euclideanGroebner:

```plaintext
a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
an:=[a1,a2,a3]
euclideanGroebner(an)
```

This will return the weak euclidean Groebner basis set. All reductions are total reductions.

You can get more information by providing a second argument. To get the reduced critical pairs do:

```plaintext
euclideanGroebner(an,"redcrit")
```
You can get other information by calling:

```
euclideanGroebner(an,"info")
```

which returns:

- `ci` => Leading monomial for critpair calculation
- `tci` => Number of terms of polynomial i
- `cj` => Leading monomial for critpair calculation
- `tcj` => Number of terms of polynomial j
- `c` => Leading monomial of critpair polynomial
- `tc` => Number of terms of critpair polynomial
- `rc` => Leading monomial of redcritpair polynomial
- `trc` => Number of terms of redcritpair polynomial
- `tH` => Number of polynomials in reduction list H
- `tD` => Number of critpairs still to do

The three argument form returns all of the information:

```
euclideanGroebner(an,"info","redcrit")
```

The term ordering is determined by the polynomial type used. Suggested types include

- `DistributedMultivariatePolynomial`
- `HomogeneousDistributedMultivariatePolynomial`
- `GeneralDistributedMultivariatePolynomial`

See Also:
- `display operations euclideanGroebner`
- `show EuclideanGroebnerBasisPackage`
- `show DistributedMultivariatePolynomial`
- `show HomogeneousDistributedMultivariatePolynomial`
- `show GeneralDistributedMultivariatePolynomial`
- `show GroebnerPackage`
EuclideanGroebnerBasisPackage (GBEUCLID)

Exports:
   euclideanGroebner   euclideanNormalForm

--- package GBEUCLID EuclideanGroebnerBasisPackage ---

)abbrev package GBEUCLID EuclideanGroebnerBasisPackage
++ Authors: Gebauer, Moeller
++ Date Created: 12-1-86
++ Date Last Updated: 2-28-91
++ Description:
++ \spadtype{EuclideanGroebnerBasisPackage} computes groebner
++ bases for polynomial ideals over euclidean domains.
++ The basic computation provides
++ a distinguished set of generators for these ideals.
++ This basis allows an easy test for membership: the operation
++ \spadfun{euclideanNormalForm} returns zero on ideal members. The string
++ "info" and "redcrit" can be given as additional args to provide
++ incremental information during the computation. If "info" is given,
++ a computational summary is given for each s-polynomial. If "redcrit"
++ is given, the reduced critical pairs are printed. The term ordering
++ is determined by the polynomial type used. Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

EuclideanGroebnerBasisPackage(Dom, Expon, VarSet, Dpol): T == C where

Dom: EuclideanDomain
Expon: OrderedAbelianMonoidSup
VarSet: OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)

T== with

   euclideanNormalForm: (Dpol, List(Dpol) ) -> Dpol
euclideanNormalForm(poly, gb) reduces the polynomial poly modulo the
precomputed groebner basis gb giving a canonical representative
of the residue class.

euclideanGroebner: List(Dpol) -> List(Dpol)
++ euclideanGroebner(lp) computes a groebner basis for a polynomial
++ ideal over a euclidean domain generated by the list of polys lp.
++
++ a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
++ a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
++ a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
++ an:=[a1,a2,a3]
++ X euclideanGroebner(an)

euclideanGroebner: (List(Dpol), String) -> List(Dpol)
++ euclideanGroebner(lp, infoflag) computes a groebner basis
++ for a polynomial ideal over a euclidean domain
++ generated by the list of polynomials lp.
++ During computation, additional information is printed out
++ if infoflag is given as
++ either "info" (for summary information) or
++ "redcrit" (for reduced critical pairs)
++
++ a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
++ a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
++ a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
++ an:=[a1,a2,a3]
++ X euclideanGroebner(an,"redcrit")
++ X euclideanGroebner(an,"info")

euclideanGroebner: (List(Dpol), String, String) -> List(Dpol)
++ euclideanGroebner(lp, "info", "redcrit") computes a groebner basis
++ for a polynomial ideal generated by the list of polynomials lp.
++ If the second argument is "info",
++ a summary is given of the critical pairs.
++ If the third argument is "redcrit", critical pairs are printed.
++
++ a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
++ a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
++ a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
++ an:=[a1,a2,a3]
++ X euclideanGroebner(an,"info","redcrit")

C== add
Ex ==> OutputForm
lc ==> leadingCoefficient
red ==> reductum

import OutputForm

------- Definition list of critPair
--- lcmfij is now lcm of headterm of poli and polj
--- lcmcij is now lcm of of lc poli and lc polj

critPair ==> Record(lcmfij: Expon, lcmcij: Dom, poli:Dpol, polj: Dpol)
Prinp ==> Record(ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol,
tc:Integer,rc:Dpol,trc:Integer,tH:Integer,tD:Integer)

--- Definition of intermediate functions

strongGbasis: (List(Dpol), Integer, Integer) -> List(Dpol)
eminGbasis: List(Dpol) -> List(Dpol)
ecritT: (critPair) -> Boolean
ecritM: (Expon, Dom, Expon, Dom) -> Boolean
ecritH: (Dpol, List(Dpol)) -> Boolean
ecritBonD: (Dpol, List(critPair)) -> List(critPair)
ecritMondd: (List(critPair)) -> List(critPair)
critdelH: (Dpol, List(Dpol)) -> List(Dpol)
eupdatF: (Dpol, List(Dpol)) -> List(Dpol)
eupdatH: (Dpol, List(Dpol), List(Dpol), List(Dpol)) -> List(Dpol)
sortin: (Dpol, List(Dpol)) -> List(Dpol)
eRed: (Dpol, List(Dpol)) -> Dpol
ecredPol: (Dpol, List(Dpol)) -> Dpol
esPol: (critPair) -> Dpol
updatD: (List(critPair), List(critPair)) -> List(critPair)
lepol: Dpol -> Integer
prinshINFO : Dpol -> Void
prindINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
prinpolINFO: List(Dpol) -> Void
prinb: Integer -> Void

--- MAIN ALGORITHM GROEBNER -------------------------
euclideanGroebner( Pol: List(Dpol) ) ==
  eminGbasis(strongGbasis(Pol,0,0))
euclideanGroebner( Pol: List(Dpol), xx1: String) ==
  xx1 = "redcrit" =>
    eminGbasis(strongGbasis(Pol,1,0))
  xx1 = "info" =>
    eminGbasis(strongGbasis(Pol,2,1))
  print("":Ex)
  print("WARNING: options are - redcrit and/or info - ":Ex)
  print("you didn't type them correct":Ex)
  print("please try again":Ex)
  print("":Ex)
[]
euclideanGroebner( Pol: List(Dpol), xx1: String, xx2: String) ==
  (xx1 = "redcrit" and xx2 = "info") or
(xx1 = "info" and xx2 = "redcrit") =>
eminGbasis(strongGbasis(Pol,1,1))
xx1 = "redcrit" and xx2 = "redcrit" =>
eminGbasis(strongGbasis(Pol,1,0))
xx1 = "info" and xx2 = "info" =>
eminGbasis(strongGbasis(Pol,2,1))
print("::Ex")
print("WARNING: options are - redcrit and/or info - "::Ex)
print("you didn't type them correct"::Ex)
print("please try again "::Ex)
print("::Ex")
[]
------ calculate basis

strongGbasis(Pol: List(Dpol),xx1: Integer, xx2: Integer ) ==
dd1, D : List(critPair)
-------- create D and Pol
Pol1:= sort((z1:Dpol,z2:Dpol):Boolean +-> (degree z1 > degree z2) or
((degree z1 = degree z2 ) and
sizeLess?(leadingCoefficient z2,leadingCoefficient z1)),
Pol)
Pol:= [first(Pol1)]
H:= Pol
Pol1:= rest(Pol1)
D:= nil
while ^null Pol1 repeat
h:= first(Pol1)
Pol1:= rest(Pol1)
en:= degree(h)
lch:= lc h
dd1:=
[[sup(degree(x), en), lcm(leadingCoefficient x, lch), x, h]$critPair
for x in Pol]
D:= updateD(
ecritMTondd1(
  ecritBonD(h,D))
Pol:= cons(h, updateF(h, Pol))
((en = degree(first(H))) and
(leadingCoefficient(h) = leadingCoefficient(first(H)) ) ) =>
" go to top of while ")
H:= updateH(h,H,crithdelH(h,H),[h])
H:= sort((z1,z2) +-> (degree z1 > degree z2) or
\[
\begin{align*}
\text{D} := \text{sort}((z_1, z_2) \rightarrow (z_1.\text{lcmfij} < z_2.\text{lcmfij}) \text{ or } \\
((z_1.\text{lcmfij} = z_2.\text{lcmfij}) \text{ and } \\
(\text{sizeLess?}(z_1.\text{lcmcij}, z_2.\text{lcmcij}))), \text{D})}
\end{align*}
\]

\[
\text{xx} := \text{xx2}
\]

\textbf{-------- loop}

\textbf{while} \neg \text{null D repeat}

\begin{align*}
\text{D0} := & \text{first} \text{D} \\
\text{ep} := & \text{esPol}(\text{D0}) \\
\text{D} := & \text{rest}(\text{D}) \\
\text{eh} := & \text{ecredPol}(\text{eRed}(\text{ep}, \text{H}), \text{H}) \\
\text{if } \text{xx1} = 1 \text{ then} \\
\text{prinshINFO(eh)} \\
\text{eh} := & 0 \rightarrow \\
\text{if } \text{xx2} = 1 \text{ then} \\
\text{ala} := & \text{prindINFO}(\text{D0}, \text{ep}, \text{H}, \#\text{D}, \text{xx}) \\
\text{xx} := & 2 \\
\text{" go to top of while "} \\
\text{eh} := & \text{unitCanonical} \text{ eh} \\
\text{e} := & \text{degree}(\text{eh}) \\
\text{leh} := & \text{lcm} \text{ eh} \\
\text{dd1} := \\
[[\text{sup}(\text{degree}(x), e), \text{lcm}(\text{leadingCoefficient} x, \text{leh}), x, \text{eh}]$\text{critPair}$ \text{ for } x \text{ in Pol}]
\end{align*}

\[
\text{D} := \text{updatD}(
\text{ecritMTondd1}(
\text{sort}((z_1, z_2) \rightarrow (z_1.\text{lcmfij} < z_2.\text{lcmfij}) \text{ or } \\
((z_1.\text{lcmfij} = z_2.\text{lcmfij}) \text{ and } \\
(\text{sizeLess?}(z_1.\text{lcmcij}, z_2.\text{lcmcij}))), \text{dd1})), \\
\text{ecritBonD}(\text{eh}, \text{D}))
\]

\[
\text{Pol} := \text{cons}(\text{eh}, \text{eupdatF}(\text{eh}, \text{Pol}))
\]

\text{ecrithinH}(\text{eh}, \text{H}) \text{ or } \text{(e = degree(first(H))) and } \\
(\text{leadingCoefficient(eh)} = \text{leadingCoefficient(first(H)))}) \Rightarrow \\
\text{if xx2 = 1 then} \\
\text{ala} := \text{prindINFO}(\text{D0}, \text{ep}, \text{H}, \#\text{D}, \text{xx}) \\
\text{xx} := 2 \\
\text{" go to top of while "} \\
\text{H} := \text{updatH}(\text{eh}, \text{H}, \text{crithdelH}(\text{eh}, \text{H}), [\text{eh}])
\]

\[
\text{H} := \text{sort}((z_1, z_2) \rightarrow \text{degree} z_1 > \text{degree} z_2) \text{ or } \\
(\text{degree} z_1 = \text{degree} z_2) \text{ and } \\
(\text{sizeLess?}(\text{leadingCoefficient z2,leadingCoefficient z1})), \text{H})
\]

\text{if xx2 = 1 then} \\
\text{ala} := \text{prindINFO}(\text{D0}, \text{ep}, \text{H}, \#\text{D}, \text{xx}) \\
\text{xx} := 2 \\
\text{" go to top of while "} \\
\text{if xx2 = 1 then}
prinpolINFO(Pol)
print(" THE GROEBNER BASIS over EUCLIDEAN DOMAIN":::Ex)
if xx1 = 1 and xx2 ^= 1 then
print(" THE GROEBNER BASIS over EUCLIDEAN DOMAIN":::Ex)
H

--------------------------------------
--- erase multiple of e in D2 using crit M
ecritMondd1(e: Expon, c: Dom, D2: List(critPair)) ==
null D2 => nil
x:= first(D2)
ecritM(e,c, x.lcmfij, lcm(leadingCoefficient(x.poli),
leadingCoefficient(x.polj)))
=> ecritMondd1(e, c, rest(D2))
cons(x, ecritMondd1(e, c, rest(D2)))

-------------------------------
ecredPol(h: Dpol, F: List(Dpol) ) ==
h0:Dpol:= 0
null F => h
while h ^= 0 repeat
  h0:= h0 + monomial(leadingCoefficient(h),degree(h))
h:= eRed(red(h), F, F)
h0
-------------------------------
--- reduce dd1 using crit T and crit M
ecritMTondd1(dd1: List(critPair)) ==
null dd1 => nil
f1:= first(dd1)
s1:= #(dd1)
cT1:= ecritT(f1)
s1= 1 and cT1 => nil
s1= 1 => dd1
e1:= f1.lcmfij
r1:= rest(dd1)
f2:= first(r1)
e1 = f2.lcmfij and f1.lcmcij = f2.lcmcij =>
cT1 => ecritMTondd1 cons(f1, rest(r1)))
ecritMTondd1(r1)
dd1 := ecritMondd1(e1, f1.lcmcij, r1)
cT1 => ecritMTondd1(dd1)
cons(f1, ecritMTondd1(dd1))

--------------------------------------
--- erase elements in D fullfilling crit B

ecritBonD(h: Dpol, D: List(critPair)) ==
null D => nil
x := first(D)
x1 := x.poli
x2 := x.polj
ecritB(degree(h), leadingCoefficient(h),
   degree(x1), leadingCoefficient(x1),
   degree(x2), leadingCoefficient(x2)) =>
ecritBonD(h, rest(D))
cons(x, ecritBonD(h, rest(D)))

--------------

--- concat F and h and erase multiples of h in F

eupdatF(h: Dpol, F: List(Dpol)) ==
null F => nil
f1 := first(F)
ecritM(degree h, leadingCoefficient(h),
   degree f1, leadingCoefficient(f1)) =>
eupdatF(h, rest(F))
cons(f1, eupdatF(h, rest(F)))

--------------

--- concat H and h and erase multiples of h in H

updatH(h: Dpol, H: List(Dpol), Hh: List(Dpol), Hhh: List(Dpol)) ==
null H => append(Hh, Hhh)
h1 := first(H)
hlcm := sup(degree(h1), degree(h))
plc := extendedEuclidean(leadingCoefficient(h), leadingCoefficient(h1))
hp := monomial(plc.coef1, subtractIfCan(hlcm, degree(h))::Expon)*h +
    monomial(plc.coef2, subtractIfCan(hlcm, degree(h1))::Expon)*h1
(ecrithinH(hp, Hh) and ecrithinH(hp, Hhh)) =>
hpp := append(rest(H), Hh)
hp := ecredPol(eRed(hp, hpp, hpp), hpp)
updatH(h, rest(H), crithdelH(hp, Hh), cons(hp, crithdelH(hp, Hhh)))
updatH(h, rest(H), Hh, Hhh)

--------------

---- delete elements in cons(h, H)

crithdelH(h: Dpol, H: List(Dpol)) ==
null H => nil
h1 := first(H)
dh1 := degree h1
dh := degree h
ecritM(dh, lc h, dh1, lc h1) => crithdelH(h, rest(H))
dh1 = sup(dh, dh1) =>
plc := extendedEuclidean(lc h1, lc h)
cons(plc.coef1*h1 + monomial(plc.coef2, subtractIfCan(dh1, dh)::Expon)*h,
    crithdelH(h, rest(H)))
cons(h1, crithdelH(h, rest(H)))

eminGbasis(F: List(Dpol)) ==
null F => nil
newbas := eminGbasis rest F
cons(ecredPol(first(F), newbas), newbas)

--- does h belong to H
ecrithinH(h: Dpol, H: List(Dpol)) ==
null H => true
h1 := first(H)
ecritM(degree h1, lc h1, degree h, lc h) => false
ecrithinH(h, rest(H))

--- calculate euclidean S-polynomial of a critical pair
esPol(p: critPair) ==
    Tij := p.lcmfij
    fi := p.poli
    fj := p.polj
    lij := lcm(leadingCoefficient(fi), leadingCoefficient(fj))
    red(fi)*monomial((lij quo leadingCoefficient(fi))::Dom,
                       subtractIfCan(Tij, degree fi)::Expon) -
    red(fj)*monomial((lij quo leadingCoefficient(fj))::Dom,
                       subtractIfCan(Tij, degree fj)::Expon)

--- euclidean reduction mod F
eRed(s: Dpol, H: List(Dpol), Hh: List(Dpol)) ==
(s = 0 or null H) => s
f1 := first(H)
ds := degree s
lf1 := leadingCoefficient(f1)
l := leadingCoefficient(s)
e: Union(Expon, "failed")
((e:= subtractIfCan(ds, degree f1)) case "failed") or sizeLess?(ls, lf1) =>
eRed(s, rest(H), Hh)
sdf1 := divide(ls, lf1)
q1 := sdf1.quotient
sdf1 remainder = 0 =>
eRed(red(s) - monomial(q1, e)*reductum(f1), Hh, Hh)
eRed(s - monomial(q1, e)*f1), rest(H), Hh)
--- crit T  true, if e1 and e2 are disjoint

ecritT(p: critPair) ==
pi:= p.poli
pj:= p.polj
ci:= lc pi
cj:= lc pj
(p.lcmFij = degree pi + degree pj) and (p.lcmCij = ci*cj)

--- crit M  true, if lcm#2 multiple of lcm#1

ecritM(e1: Expon, c1: Dom, e2: Expon, c2: Dom) ==
en: Union(Expon, "failed")
((en:=subtractIfCan(e2, e1)) case "failed") or
((c2 exquo c1) case "failed") => false
true

--- crit B  true, if eik is a multiple of eh and eik ^equal
--- lcm(eh,ei) and eik ^equal lcm(eh,ek)

eik:= sup(ei, ek)
cik:= lcm(ci, ck)
ecritM(eh, ch, eik, cik) and
^ecritM(eik, cik, sup(ei, eh), lcm(ci, ch)) and
^ecritM(eik, cik, sup(ek, eh), lcm(ck, ch))

--- reduce p1 mod lp

euclideanNormalForm(p1: Dpol, lp: List(Dpol))==
eRed(p1, lp, lp)

--- insert element in sorted list

sortin(p1: Dpol, lp: List(Dpol))==
null lp => [p1]
f1:= first(lp)
elf1:= degree(f1)
ep1:= degree(p1)
((elf1 < ep1) or ((elf1 = ep1) and
sizeLess?(leadingCoefficient(f1),leadingCoefficient(p1))) =>
    cons(f1,sortin(p1, rest(lp)))
cons(p1,lp)

updatD(D1: List(critPair), D2: List(critPair)) ==
    null D1 => D2
    null D2 => D1
    dl1:= first(D1)
    dl2:= first(D2)
    (dl1.lcmfij < dl2.lcmfij) => cons(dl1, updatD(D1.rest, D2))
    cons(dl2, updatD(D1, D2.rest))

    ---- calculate number of terms of polynomial

lepol(p1:Dpol)==
    n: Integer
    n:= 0
    while p1 ^= 0 repeat
        n:= n + 1
        p1:= red(p1)
    n

    ---- print blanc lines

prinb(n: Integer)==
    for i in 1..n repeat messagePrint(" ")

    ---- print reduced critpair polynom

prinshINFO(h: Dpol)==
    prinb(2)
    messagePrint(" reduced Critpair - Polynom :")
    prinb(2)
    print(h::Ex)
    prinb(2)

-----------------------------

    ---- print info string

prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer, i2:Integer, n:Integer) ==
    ll: List Prinp
    a: Dom
    cpi:= cp.poli
    cpj:= cp.polj
    if n = 1 then
        prinb(1)
        messagePrint("you choose option -info- ")
        messagePrint("abbrev. for the following information strings are")
messagePrint(" ci => Leading monomial for critpair calculation")
messagePrint(" tci => Number of terms of polynomial i")
messagePrint(" cj => Leading monomial for critpair calculation")
messagePrint(" tcj => Number of terms of polynomial j")
messagePrint(" c => Leading monomial of critpair polynomial")
messagePrint(" tc => Number of terms of critpair polynomial")
messagePrint(" rc => Leading monomial of redcritpair polynomial")
messagePrint(" trc => Number of terms of redcritpair polynomial")
messagePrint(" tF => Number of polynomials in reduction list F")
messagePrint(" tD => Number of critpairs still to do")
prinb(4)
n:= 2
prinb(1)
a:= 1
ph = 0 =>
ps = 0 =>
ll:= [[monomial(a,degree(cpi)),lepol(cpi),monomial(a,degree(cpj)),
  lepol(cpj),ps,0,ph,0,i1,i2]$Prinp]
print(ll::Ex)
prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
  monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
  lepol(ps), ph,0,i1,i2]$Prinp]
print(ll::Ex)
prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
  monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
  lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2]$Prinp]
print(ll::Ex)
prinb(1)
n
-------------------------------
---- print the groebner basis polynomials

prinpolINFO(pl: List(Dpol))==
n:Integer
n:= #pl
prinb(1)
n = 1 =>
  print(" There is 1 Groebner Basis Polynomial ":Ex)
  prinb(2)
print(" There are ":Ex)
prinb(1)
print(n::Ex)
prinb(1)
print(" Groebner Basis Polynomials. ":Ex)
package EVALCYC EvaluateCycleIndicators

--- EvaluateCycleIndicators.input ---

)set break resume
)sys rm -f EvaluateCycleIndicators.output
)spool EvaluateCycleIndicators.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EvaluateCycleIndicators
--E 1

)spool
)lisp (bye)

--- EvaluateCycleIndicators.help ---

====================================================================
EvaluateCycleIndicators examples
====================================================================

This package is to be used in conjunction with the CycleIndicators package. It provides an evaluation function for SymmetricPolynomials.
See Also:
- )show EvaluateCycleIndicators

EvaluateCycleIndicators (EVALCYC)

Exports:
- eval

--- package EVALCYC EvaluateCycleIndicators ---

EvaluateCycleIndicators(F):T==C where
  F: Algebra Fraction Integer
  I==>Integer
  L==>List
  SPOL==>SymmetricPolynomial
  RN==>Fraction Integer
  PR==>Polynomial(RN)
  PTN==>Partition()
  lc ==> leadingCoefficient
  red ==> reductum
  T== with
    eval:((I->F),SPOL RN)->F
++\spad{eval(f,s)} evaluates the cycle index s by applying ++ the function f to each integer in a monomial partition, ++ forms their product and sums the results over all monomials.

\begin{verbatim}
C== add evp:((I->F),PTN)->F fn:I->F pt:PTN spol:SPOL RN i:I evp(fn, pt)== */[fn i for i in pt::(L I)]
eval(fn,spol)== if spol=0 then 0 else ((lc spol)* evp(fn,degree spol)) + eval(fn,red spol)
\end{verbatim}

--- EVALCYC.dotabb ---

"EVALCYC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EVALCYC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"EVALCYC" -> "PFECAT"

---

package ESCONT ExpertSystemContinuityPackage

--- ExpertSystemContinuityPackage.input ---

)set break resume
)sys rm -f ExpertSystemContinuityPackage.output
)spool ExpertSystemContinuityPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ExpertSystemContinuityPackage
--E 1

)spool
)lisp (bye)
ExpertSystemContinuityPackage (ESCONT)

Exports:

- df2st
- functionIsFracPolynomial?
- gethi
- getlo
- ldf2lst
- polynomialZeros
- problemPoints
- sdf2lst
- singularitiesOf
- zerosOf

--- package ESCONT ExpertSystemContinuityPackage ---

)abbrev package ESCONT ExpertSystemContinuityPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
++ Description:
++ ExpertSystemContinuityPackage is a package of functions for the use of
++ domains belonging to the category \axiomType{NumericalIntegration}. 
ExpertSystemContinuityPackage(): E == I where

EF2  ==> ExpressionFunctions2
FI   ==> Fraction Integer
EFI  ==> Expression Fraction Integer
PFI  ==> Polynomial Fraction Integer
DF   ==> DoubleFloat
LDF  ==> List DoubleFloat
EDF  ==> Expression DoubleFloat
VEDF ==> Vector Expression DoubleFloat
SDF  ==> Stream DoubleFloat
SS   ==> Stream String
EEDF ==> Equation Expression DoubleFloat
LEDF ==> List Expression DoubleFloat
KEDF ==> Kernel Expression DoubleFloat
LKEDF==> List Kernel Expression DoubleFloat
PDF  ==> Polynomial DoubleFloat
FPDF ==> Fraction Polynomial DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
SOCDF==> Segment OrderedCompletion DoubleFloat
NIA  ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
UP   ==> UnivariatePolynomial
BO   ==> BasicOperator
RS   ==> Record(zeros: SDF,ones: SDF,singularities: SDF)

E ==>

getlo : SOCDF -> DF
++ getlo(u) gets the \axiomType{DoubleFloat} equivalent of
++ the first endpoint of the range \axiom{u}

gethi : SOCDF -> DF
++ gethi(u) gets the \axiomType{DoubleFloat} equivalent of
++ the second endpoint of the range \axiom{u}

functionIsFracPolynomial?: NIA -> Boolean
++ functionIsFracPolynomial?(args) tests whether the function
++ can be retracted to \axiomType{Fraction(Polynomial(DoubleFloat))}

problemPoints:(EDF,Symbol,SOCDF) -> List DF
++ problemPoints(f,var,range) returns a list of possible problem points
++ by looking at the zeros of the denominator of the function \spad{f}
++ if it can be retracted to \axiomType{Polynomial(DoubleFloat)}.

zerosOf:(EDF,List Symbol,SOCDF) -> SDF
++ zerosOf(e,vars,range) returns a list of points
++ (\axiomType{Doublefloat}) at which a NAG fortran version of \spad{e}
++ will most likely produce an error.

singularitiesOf: (EDF,List Symbol,SOCDF) -> SDF
++ singularitiesOf(e,vars,range) returns a list of points
++ (\axiomType{Doublefloat}) at which a NAG fortran
++ version of \spad{e} will most likely produce
++ an error. This includes those points which evaluate to 0/0.

singularitiesOf: (Vector EDF,List Symbol,SOCDF) -> SDF
++ singularitiesOf(v,vars,range) returns a list of points
++ at which a NAG fortran ++ version of \spad{v} will most likely produce ++ an error. This includes those points which evaluate to 0/0.

polynomialZeros:(PFI,Symbol,SOCDF) -> LDF ++ polynomialZeros(fn,\var,range) calculates the real zeros of the ++ polynomial which are contained in the given interval. It returns ++ a list of points \axiomType{Doublefloat} for which the univariate ++ polynomial \spad{fn} is zero.

def2st:DF -> String ++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}

ldf2lst:LDF -> List String ++ ldf2lst(ln) coerces a List of \axiomType{DoubleFloat} to ++ \axiomType{List}(\axiomType{String})

 sdf2lst:SDF -> List String ++ sdf2lst(ln) coerces a Stream of \axiomType{DoubleFloat} to ++ \axiomType{List}(\axiomType{String})

I ==> ExpertSystemToolsPackage add

import ExpertSystemToolsPackage

functionIsPolynomial?(args:NIA):Boolean ==
-- tests whether the function can be retracted to a polynomial (retractIfCan(args.fn)@Union(PDF,"failed"))$EDF case PDF

isPolynomial?(f:EDF):Boolean ==
-- tests whether the function can be retracted to a polynomial (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

isConstant?(f:EDF):Boolean ==
-- tests whether the function can be retracted to a constant (DoubleFloat) (retractIfCan(f)@Union(DF,"failed"))$EDF case DF

denominatorIsPolynomial?(args:NIA):Boolean ==
-- tests if the denominator can be retracted to polynomial a:= copy args a.fn:=denominator(args.fn) (functionIsPolynomial?(a))@Boolean

denIsPolynomial?(f:EDF):Boolean ==
-- tests if the denominator can be retracted to polynomial (isPolynomial?(denominator f))@Boolean

listInRange(l:LDF,range:SOCDF):LDF ==
-- returns a list with only those elements internal to the range range [t for t in l | in?(t,range)]

loseUntil(l:SDF,a:DF):SDF ==
  empty?(l)$SDF => l
  f := first(l)$SDF
(abs(f) <= abs(a)) => loseUntil(rest(l)$SDF,a)

retainUntil(l:SDF,a:DF,b:DF,flag:Boolean):SDF ==
  empty?(l)$SDF => l
  f := first(l)$SDF
  (in?(f)$ExpertSystemContinuityPackage1(a,b)) =>
    concat(f,retainUntil(rest(l),a,b,false))
  flag => empty()$SDF
  retainUntil(rest(l),a,b,true)

streamInRange(l:SDF,range:SOCDF):SDF ==
  -- returns a stream with only those elements internal to the range range
  a := getlo(range := dfRange(range))
  b := gethi(range)
  explicitlyFinite?(l) =>
    select(in?$ExpertSystemContinuityPackage1(a,b),l)$SDF
  negative?(a*b) => retainUntil(l,a,b,false)
  negative?(a) =>
    l := loseUntil(l,b)
    retainUntil(l,a,b,false)
  l := loseUntil(l,a)
  retainUntil(l,a,b,false)

getStream(n:Symbol,s:String):SDF ==
  import RS
  entry?(n,bfKeys()$BasicFunctions)$(List(Symbol)) =>
    c := bfEntry(n)$BasicFunctions
    (s = "zeros")@Boolean => c.zeros
    (s = "singularities")@Boolean => c.singularities
    (s = "ones")@Boolean => c.ones
    empty()$SDF

polynomialZeros(fn:PFI,var:Symbol,range:SOCDF):LDF ==
  up := unmakeSUP(univariate(fn)$PFI)$UP(var,FI)
  range := dfRange(range)
  r:Record(left:FI,right:FI) := [df2fi(getlo(range)), df2fi(gethi(range))]
  ans:List(Record(left:FI,right:FI)) :=
    realZeros(up,r,1/1000000000000000000)$RealZeroPackageQ(UP(var,FI))
    listInRange(dflist(ans),range)

functionIsFracPolynomial?(args:NIA):Boolean ==
  -- tests whether the function can be retracted to a fraction
  -- where both numerator and denominator are polynomial
  (retractIfCan(args.fn)@Union(FPDF,"failed"))$EDF case FPDF

problemPoints(f:EDF,var:Symbol,range:SOCDF):LDF ==
  (denIsPolynomial?(f))@Boolean =>
    c := retract(edf2efi(denominator(f)))@PFI
    polynomialZeros(c,var,range)
empty()$LDF

zerosOf(e:EDF,vars:List Symbol,range:SCDF):SDF ==
(u := isQuotient(e)) case EDF =>
singularitiesOf(u,vars,range)
k := kernels(e)$EDF
((nk := # k) = 0)$Boolean => empty()$SDF -- constant found.
(nk = 1)$Boolean => -- single expression found.
k := first(k)$LKEDF
n := name(operator(ker)$KEDF)$BO
entry?(n,vars) => -- polynomial found.
c := retract(edf2efi(e))@PFI
coerce(polynomialZeros(c,n,range))$SDF
a := first(argument(ker)$KEDF)$LEDF
(not (n = log :: Symbol)$Boolean) and (((w := isPlus a) case LEDF) =>
var:Symbol := first(variables(a))
c:EDF := w.2
c1:EDF := w.1
-- entry?(c1,[b::EDF for b in vars]) and (one?(# vars)) =>
entry?(c1,[b::EDF for b in vars]) and ((# vars) = 1) =>
c2:DF := edf2df c
C3 := c2 :: OCDF
varEdf := var :: EDF
varEqn := equation(varEdf,c1-c)$EEDF
range2 := (lo(range)+c3)..(hi(range)+c3)
s := zerosOf(subst(e,varEqn)$EDF,vars,range2)
st := map(t1 +-> t1-c2,s)$StreamFunctions2(DF,DF)
streamInRange(st,range)
zerosOf(a,vars,range)
(t := isPlus(e)$EDF) case LEDF => -- constant + expression
# t > 2 => empty()$SDF
entry?(a,[b::EDF for b in vars]) => -- finds entries like sqrt(x)
st := getStream(n,"ones")
o := edf2df(second(t)$LEDF)
-- one?(o) or one?(-o) => -- is it like (f(x) -/+ 1)
(o = 1) or (-o = 1) =>
-- is it like (f(x) -/+ 1)
st := map(t2 +-> -t2/o,st)$StreamFunctions2(DF,DF)
streamInRange(st,range)
empty()$SDF
empty()$SDF
entry?(a,[b::EDF for b in vars]) => -- finds entries like sqrt(x)
st := getStream(n,"zeros")
streamInRange(st,range)
(n = tan :: Symbol)$Boolean =>
concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
(n = sin :: Symbol)$Boolean =>
concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
empty()$SDF
(t := isPlus(e)$EDF) case LEDF => empty()$SDF -- INCOMPLETE!!
(v := isTimes(e)$EDF) case LEDF =>
concat([zerosOf(u,vars,range) for u in v])
empty()$SDF

singularitiesOf(e:EDF,vars:List Symbol,range:SOCDF):SDF ==
(u := isQuotient(e)) case EDF => zerosOf(u,vars,range)
(t := isPlus e) case LEDF => concat([singularitiesOf(u,vars,range) for u in t])
(v := isTimes e) case LEDF => concat([singularitiesOf(u,vars,range) for u in v])
(k := mainKernel e) case KEDF =>
  n := name(operator k)
  entry?(n,vars) => coerce(problemPoints(e,n,range))$SDF
  a:EDF := (argument k).1
  (not (n = log :: Symbol)$Boolean) and ((w := isPlus a) case LEDF) =>
    varSymbol := first(variables(a))
    c:EDF := w.2
    c1:EDF := w.1
    entry?(c1,[b::EDF for b in vars]) and (one?(# vars)) =>
    entry?(c1,[b::EDF for b in vars]) and ((# vars) = 1) =>
      c2:DF := edf2df c
      c3 := c2 :: OCDF
      varEdf := var :: EDF
      varEqn := equation(varEdf,c1-c)$EEDF
      range2 := (lo(range)+c3)..(hi(range)+c3)
      s := singularitiesOf(subst(e,varEqn)$EDF,vars,range2)
      st := map(t3 +-> t3-c2,s)$StreamFunctions2(DF,DF)
      streamInRange(st,range)
    singularitiesOf(a,vars,range)
  entry?(a,[b::EDF for b in vars]) =>
    entry?(a,[b::EDF for b in vars]) and (one?(# vars)) =>
    concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
    singularitiesOf(a,vars,range)
package ESCONT1 ExpertSystemContinuityPackage1

---

ExpertSystemContinuityPackage1 exports a function to check range inclusion

See Also:
- )show ExpertSystemContinuityPackage1
ExpertSystemContinuityPackage1 (ESCONT1)

Exports:
in?

— package ESCONT1 ExpertSystemContinuityPackage1 —

)abbrev package ESCONT1 ExpertSystemContinuityPackage1
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
++ Description:
++ ExpertSystemContinuityPackage1 exports a function to check range inclusion

ExpertSystemContinuityPackage1(A:DF,B:DF): E == I where
  EF2 ==> ExpressionFunctions2
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  PFI ==> Polynomial Fraction Integer
  DF ==> DoubleFloat
  LDF ==> List DoubleFloat
  EDF ==> Expression DoubleFloat
  VEDF ==> Vector Expression DoubleFloat
  SDF ==> Stream DoubleFloat
  SS ==> Stream String
  EEDF ==> Equation Expression DoubleFloat
  LEDF ==> List Expression DoubleFloat
  KEDF ==> Kernel Expression DoubleFloat
  LKEDF ==> List Kernel Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  FPDF ==> Fraction Polynomial DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  UP ==> UnivariatePolynomial
  BO ==> BasicOperator
  RS ==> Record(zeros: SDF,ones: SDF,singularities: SDF)
E ==> with

  in?:DF -> Boolean
   ++ in?(p) tests whether point p is internal to the range \(\text{\texttt{A..B}}\)

I ==> add

  in?(p:DF):Boolean ==
    a:Boolean := (p < B)$DF
    b:Boolean := (A < p)$DF
    (a and b)$Boolean

--

--- ESCONT1.dotabb ---

"ESCONT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESCONT1"]
"Package" [color="#FF4488"]
"ESCONT1" -> "Package"

---

package ESTOOLS ExpertSystemToolsPackage

--- ExpertSystemToolsPackage.input ---

)set break resume
)sys rm -f ExpertSystemToolsPackage.output
)spool ExpertSystemToolsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpertSystemToolsPackage
--E 1

)spool
)lisp (bye)

---

--- ExpertSystemToolsPackage.help ---
ExpertSystemToolsPackage examples

ExpertSystemToolsPackage contains some useful functions for use by the computational agents of numerical solvers.

See Also:
- \show ExpertSystemToolsPackage

---

\textbf{ExpertSystemToolsPackage (ESTOOLS)}

\begin{itemize}
\item \textbf{Exports:}
  \begin{itemize}
  \item att2Result
  \item df2ef
  \item edf2ef
  \item f2df
  \item iflist2Result
  \item mat
  \item outputMeasure
  \item vef2vef
  \item concat
  \item convert
  \item df2ef
  \item edf2ef
  \item f2df
  \item in?
  \item measure2Result
  \item pdf2df
  \item pdf2ef
  \item dflist
  \item dfst
  \item edf2mf
  \item edf2ef
  \item f2st
  \item gethi
  \item ldf2lst
  \item measure2Result
  \item numberOfOperations
  \item ocf2ocdf
  \item pdf2df
  \item pdf2ef
  \item sdf2lst
  \item expenseOfEvaluation
  \item getlo
  \item ld2vmf
  \item mat
  \item outputMeasure
  \item vef2vef
  \end{itemize}
\end{itemize}

\texttt{— package ESTOOLS ExpertSystemToolsPackage —}

\texttt{)}abbrev package ESTOOLS ExpertSystemToolsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: July 1996
++ Description:
++ \texttt{axiom\{ExpertSystemToolsPackage\}} contains some useful functions for use
++ by the computational agents of numerical solvers.
ExpertSystemToolsPackage():E == I where
LEDF ==> List Expression DoubleFloat
KEDF ==> Kernel Expression DoubleFloat
LKEDF ==> List Kernel Expression DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
VMF ==> Vector MachineFloat
EF2 ==> ExpressionFunctions2
EFI ==> Expression Fraction Integer
MDF ==> Matrix DoubleFloat
LDF ==> List DoubleFloat
PDF ==> Polynomial DoubleFloat
EDF ==> Expression DoubleFloat
EF ==> Expression Float
SDF ==> Stream DoubleFloat
DF ==> DoubleFloat
F ==> Float
MF ==> MachineFloat
INT ==> Integer
NNI ==> NonNegativeInteger
LS ==> List Symbol
ST ==> String
LST ==> List String
SS ==> Stream String
FI ==> Fraction Integer
R ==> Ring
OR ==> OrderedRing
ON ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
RVE ==> Record(val:EDF,exponent:INT)
BO ==> BasicOperator
OCF ==> OrderedCompletion Float
OCDF ==> OrderedCompletion DoubleFloat
SOCF ==> Segment OrderedCompletion Float
SOCDF ==> Segment OrderedCompletion DoubleFloat
Measure ==> Record(measure:F, name:String, explanations:List String)
Measure2 ==> Record(measure:F, name:String, explanations:List String, extra:Result)
CTYPE ==> Union(continuous: "Continuous at the end points",
lowerSingular: "There is a singularity at the lower end point",
upperSingular: "There is a singularity at the upper end point",
bothSingular: "There are singularities at both end points",
notevaluated: "End point continuity not yet evaluated")
RTYPE ==> Union(finite: "The range is finite",
lowerInfinite: "The bottom of range is infinite",
upperInfinite: "The top of range is infinite",
bothInfinite: "Both top and bottom points are infinite",
notevaluated: "Range not yet evaluated")
STYPE ==> Union(str:SDF,
notevaluated: "Internal singularities not yet evaluated")
ATT ==> Record(endPointContinuity:CTYPE,singularitiesStream:STYPE,range:RTYPE)

E ==> with

f2df:F -> DF
++ f2df(f) is a function to convert a \axiomType{Float} to a
++ \axiomType{DoubleFloat}

ef2edf:EF -> EDF
++ ef2edf(f) is a function to convert an \axiomType{Expression Float}
++ to an \axiomType{Expression DoubleFloat}

ocf2ocdf: OCF -> OCDF
++ ocf2ocdf(a) is a function to convert an \axiomType{OrderedCompletion
++ Float} to an \axiomType{OrderedCompletion DoubleFloat}

socf2socdf: SOCF -> SOCDF
++ socf2socdf(a) is a function to convert a \axiomType{Segment OrderedCompletion Float}
++ to a \axiomType{Segment OrderedCompletion DoubleFloat}

convert: List SOCF -> List SOCDF
++ convert(l) is a function to convert a \axiomType{Segment OrderedCompletion Float}
++ to a \axiomType{Segment OrderedCompletion DoubleFloat}

df2fi :DF -> FI
++ df2fi(n) is a function to convert a \axiomType{DoubleFloat} to a
++ \axiomType{Fraction Integer}

edf2fi :EDF -> FI
++ edf2fi(n) maps \axiomType{Expression DoubleFloat} to
++ \axiomType{Fraction Integer}
++ It is an error if \spad{n} is not coercible to \axiomType{Fraction Integer}

edf2df :EDF -> DF
++ edf2df(n) maps \axiomType{Expression DoubleFloat} to
++ \axiomType{DoubleFloat}
++ It is an error if \spad{n} is not coercible to \axiomType{DoubleFloat}

isQuotient: EDF -> Union(EDF,"failed")
++ isQuotient(expr) returns the quotient part of the input
++ expression or \spad{"failed"} if the expression is not of that form.

expenseOfEvaluation: VEDF -> F
++ expenseOfEvaluation(o) gives an approximation of the cost of
++ evaluating a list of expressions in terms of the number of basic
++ operations.
++ < 0.3 inexpensive ; 0.5 neutral ; > 0.7 very expensive
++ 400 'operation units' -> 0.75
++ 200 'operation units' -> 0.5
++ 83 'operation units' -> 0.25
++ ** = 4 units , function calls = 10 units.

numberOfOperations: VEDF -> ON
++ numberOfOperations(ode) counts additions, multiplications,
++ exponentiations and function calls in the input set of expressions.

df2efi: EDF -> EFI
++ df2efi(e) coerces \axiomType{Expression DoubleFloat} into
++ \axiomType{Expression Fraction Integer}

dfRange: SOCDF -> SOCDF
++ dfRange(r) converts a range including
++ \inputbitmap{\htbmdir{}/plusminus.bitmap} \infty
++ to \axiomType{DoubleFloat} equivalents.
dflist:List(Record(left:FI,right:FI)) -> LDF
++ dflist(l) returns a list of \axiomType{DoubleFloat} equivalents of list l
df2mf:DF -> MF
++ df2mf(n) coerces a \axiomType{DoubleFloat} to \axiomType{MachineFloat}
1df2vmf:LDF -> VMF
++ 1df2vmf(l) coerces a \axiomType{List DoubleFloat} to
++ \axiomType{List MachineFloat}
edf2ef:EDF -> EF
++ edf2ef(e) maps \axiomType{Expression DoubleFloat} to
++ \axiomType{Expression Float}
vedf2vef:VEDF -> VEF
++ vedf2vef(v) maps \axiomType{Vector Expression DoubleFloat} to
++ \axiomType{Vector Expression Float}
in?:DF,SOCDF -> Boolean
++ in?(p,range) tests whether point p is internal to the
++ range range
df2st:DF -> ST
++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
f2st:F -> ST
++ f2st(n) coerces a \axiomType{Float} to \axiomType{String}
1df2lst:LDF -> LST
++ 1df2lst(ln) coerces a \axiomType{List DoubleFloat} to \axiomType{List String}
sdf2lst:SDF -> LST
++ sdf2lst(ln) coerces a \axiomType{Stream DoubleFloat} to \axiomType{String}
getlo : SOCDF -> DF
++ getlo(u) gets the \axiomType{DoubleFloat} equivalent of
++ the first endpoint of the range \spad{u}
gethi : SOCDF -> DF
++ gethi(u) gets the \axiomType{DoubleFloat} equivalent of
++ the second endpoint of the range \spad{u}
concat:(Result,Result) -> Result
++ concat(a,b) adds two aggregates of type \axiomType{Result}.
concat:(List Result) -> Result
++ concat(l) concatenates a list of aggregates of type \axiomType{Result}
outputMeasure:F -> ST
++ outputMeasure(n) rounds \spad{n} to 3 decimal places and outputs
++ it as a string
measure2Result:Measure -> Result
++ measure2Result(m) converts a measure record into a \axiomType{Result}
measure2Result:Measure2 -> Result
++ measure2Result(m) converts a measure record into a \axiomType{Result}
att2Result:ATT -> Result
++ att2Result(m) converts a attributes record into a \axiomType{Result}
iflist2Result:IFV -> Result
++ iflist2Result(m) converts a attributes record into a \axiomType{Result}
pdf2ef:PDF -> EF
++ pdf2ef(p) coerces a \axiomType{Polynomial DoubleFloat} to
++ \axiomType{Expression Float}
pdf2df:PDF -> DF
++ pdf2df(p) coerces a \axiomType{Polynomial DoubleFloat} to
++ \axiomType{DoubleFloat}. It is an error if \axiom{p} is not
++ retractable to DoubleFloat.
df2ef:DF -> EF
++ df2ef(a) coerces a \axiomType{DoubleFloat} to \axiomType{Expression Float}
fi2df:FI -> DF
++ fi2df(f) coerces a \axiomType{Fraction Integer} to \axiomType{DoubleFloat}
mat:(LDF,NNI) -> MDF
++ mat(a,n) constructs a one-dimensional matrix of a.

I ==> add

mat(a:LDF,n:NNI):MDF ==
   empty?(a)$LDF => zero(1,n)$MDF
   matrix(list([i for i in a for j in 1..n])$(List LDF))$MDF

f2df(f:F):DF == (convert(f)@DF)$F
ef2ef(f:EF):EDF == map(f2df,f)$EF2(F,DF)
fi2df(f:FI):DF == coerce(f)$DF
ocf2ocdf(a:OCF):OCDF ==
   finite? a => (f2df(retract(a)$F))::OCDF
   a pretend OCDF
socf2socdf(a:SOCF):SOCDF ==
   segment(ocf2ocdf(lo a),ocf2ocdf(hi a))
convert(l:List SOCF):List SOCDF == [socf2socdf a for a in l]

pdf2df(p:PDF):DF == retract(p)$DF

df2ef(a:DF):EF ==
   b := convert(a)$Float
   coerce(b)$EF

df2ef(p:PDF):EF == df2ef(pdf2df(p))
edf2fi(m:EDF):FI == retract(retract(m)$DF)$FI

edf2df(m:EDF):DF == retract(m)$DF

df2fi(r:DF):FI == (retract(r)$FI)$DF

dfRange(r:SOCDF):SOCDF ==
   if infinite?(lo(r))$OCDF then r := -(max()$DF :: OCDF)..hi(r)$SOCDF
   if infinite?(hi(r))$OCDF then r := lo(r)$SOCDF..(max()$DF :: OCDF)
   r
dflist(l: List(Record(left: FI, right: FI))): LDF == [u.left :: DF for u in l]

edf2efi(f: EDF): EFI == map(df2fi, f)$EF2(DF, FI)

df2st(n: DF): String == (convert((convert(n)@Float)$DF)@ST)$Float

f2st(n: F): String == (convert(n)@ST)$Float

ldf2lst(ln: LDF): LST == [df2st f for f in ln]

sdf2lst(ln: SDF): LST ==
  explicitlyFinite? ln =>
    m := map(df2st, ln)$StreamFunctions2(DF, ST)
    if index?(20, m)$SS then
      split!(m, 20)
    m := concat(m, "........")
    entries(m)$SS
  empty()$LST

df2mf(n: DF): MF == (df2fi(n))::MF

ldf2vmf(l: LDF): VMF ==
  m := [df2mf(n) for n in l]
  vector(m)$VMF

edf2ef(e: EDF): EF == map(convert$DF, e)$EF2(DF, Float)

vedf2vef(vedf: VEDF): VEF == vector([edf2ef e for e in members(vedf)])

getlo(u: SOCDF): DF == retract(lo(u))@DF

gethi(u: SOCDF): DF == retract(hi(u))@DF

in?(p: DF, range: SOCDF): Boolean ==
  top := gethi(range)
  bottom := getlo(range)
  a:Boolean := (p < top)$DF
  b:Boolean := (p > bottom)$DF
  (a and b)$Boolean

isQuotient(expr: EDF): Union(EDF, "failed") ==
  (k := mainKernel expr) case KEDF =>
    (expr = inv(f := k :: KEDF :: EDF)$EDF)$EDF => f
  one?(numerator expr) => denominator expr
  (numerator expr) = 1 => denominator expr
  "failed"
  "failed"
CHAPTER 6. CHAPTER E

numberOfOperations1(fn: EDF, numbersSoFar: ON) == 
  (u := isQuotient(fn)) case EDF =>
      numbersSoFar := numberOfOperations1(u, numbersSoFar)
  (p := isPlus(fn)) case LEDF =>
      p := coerce(p)@LEDF
      np := #p
      numbersSoFar.additions := (numbersSoFar.additions)+np-1
      for i in 1..np repeat
          numbersSoFar := numberOfOperations1(p.i, numbersSoFar)
      numbersSoFar
  (t := isTimes(fn)) case LEDF =>
      t := coerce(t)@LEDF
      nt := #t
      numbersSoFar.multiplications := (numbersSoFar.multiplications)+nt-1
      for i in 1..nt repeat
          numbersSoFar := numberOfOperations1(t.i, numbersSoFar)
      numbersSoFar
  if (e := isPower(fn)) case RVE then
      e := coerce(e)@RVE
      e.exponent>1 =>
          numbersSoFar.exponentiations := inc(numbersSoFar.exponentiations)
      numbersSoFar := numberOfOperations1(e.val, numbersSoFar)
  lk := kernels(fn)
  #lk = 1 => -- #lk = 0 => constant found (no further action)
  k := first(lk)$LKEDF
  n := name(operator(k)$KEDF)$BO
  entry?(n, variables(fn)$EDF)$LS => numbersSoFar -- solo variable found
  a := first(argument(k)$KEDF)$LEDF
  numbersSoFar.functionCalls := inc(numbersSoFar.functionCalls)$INT
  numbersSoFar := numberOfOperations1(a, numbersSoFar)
numbersSoFar

numberOfOperations(ode: VEDF): ON ==
  n: ON := [0, 0, 0, 0]
  for i in 1..#ode repeat
      n: ON := numberOfOperations1(ode.i, n)
  n

expenseOfEvaluation(o: VEDF): F ==
  ln: ON := numberOfOperations(o)
  a := ln.additions
  m := ln.multiplications
  e := ln.exponentiations
  f := 10*ln.functionCalls
  n := (a + m + 4*e + 10*e)
  (1.0-exp((-n:: F/288.0))$F)

concat(a: Result, b: Result): Result ==
  membersOfa := (members(a)@List(Record(key: Symbol, entry: Any)))
  membersOfb := (members(b)@List(Record(key: Symbol, entry: Any)))
allMembers :=
    concat(membersOfa,membersOfb)$List(Record(key:Symbol,entry:Any))
construct(allMembers)

concat(l:List Result):Result ==
    import List Result
    empty? l => empty()$Result
    f := first l
    if empty?(r := rest l) then
        f
    else
        concat(f,concat r)

outputMeasure(m:F):ST ==
    f1:Float := round(m*(f:= 1000.0))/f
    convert(f1)$ST

measure2Result(m:Measure):Result ==
    mm := coerce(m.measure)$AnyFunctions1(Float)
    mnr:Record(key:Symbol,entry:Any) := [bestMeasure@Symbol,mm]
    mn := coerce(m.name)$AnyFunctions1(ST)
    mnr:Record(key:Symbol,entry:Any) := [nameOfRoutine@Symbol,mn]
    me := coerce(m.explanations)$AnyFunctions1(List String)
    mer:Record(key:Symbol,entry:Any) := [allMeasures@Symbol,me]
    mr := construct([mmr,mnr,mer])$Result
    met := coerce(mr)$AnyFunctions1(Result)
    meth:Record(key:Symbol,entry:Any):=[method@Symbol,met]
    construct([meth])$Result

measure2Result(m:Measure2):Result ==
    mm := coerce(m.measure)$AnyFunctions1(Float)
    mnr:Record(key:Symbol,entry:Any) := [bestMeasure@Symbol,mm]
    mn := coerce(m.name)$AnyFunctions1(ST)
    mnr:Record(key:Symbol,entry:Any) := [nameOfRoutine@Symbol,mn]
    me := coerce(m.explanations)$AnyFunctions1(List String)
    mer:Record(key:Symbol,entry:Any) := [allMeasures@Symbol,me]
    mx := coerce(m.extra)$AnyFunctions1(Result)
    mxr:Record(key:Symbol,entry:Any) := [other@Symbol,mx]
    mr := construct([mmr,mnr,mer,mxr])$Result
    met := coerce(mr)$AnyFunctions1(Result)
    meth:Record(key:Symbol,entry:Any):=[method@Symbol,met]
    construct([meth])$Result

att2Result(att:ATT):Result ==
    aepc := coerce(att.endPointContinuity)$AnyFunctions1(CTYPE)
    ar := coerce(att.range)$AnyFunctions1(RTYPE)
    as := coerce(att.singularitiesStream)$AnyFunctions1(STYPE)
    aa:List Any := [aepc,ar,as]
    aaa := coerce(aa)$AnyFunctions1(List Any)
    aar:Record(key:Symbol,entry:Any) := [attributes@Symbol,aaa]
begin

construct([aar])$Result

iflist2Result(ifv:IFV):Result ==
ifvs:List String :=
[concat(["stiffness: ",outputMeasure(ifv.stiffness)]),
 concat(["stability: ",outputMeasure(ifv.stability)]),
 concat(["expense: ",outputMeasure(ifv.expense)]),
 concat(["accuracy: ",outputMeasure(ifv.accuracy)]),
 concat(["intermediateResults: ",outputMeasure(ifv.intermediateResults)]])
ifa:= coerce(ifvs)$AnyFunctions1(List String)
ifr:Record(key:Symbol,entry:Any) := [intensityFunctions@Symbol,ifa]
construct([ifr])$Result

——

ESTOOLS.dotabb —

"ESTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ESTOOLS" -> "FS"

——

package ESTOOLS1 ExpertSystemToolsPackage1

——

ExpertSystemToolsPackage1.input —

)set break resume
)sys rm -f ExpertSystemToolsPackage1.output
)spool ExpertSystemToolsPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpertSystemToolsPackage1
--E 1

)spool
)lisp (bye)

——

ExpertSystemToolsPackage1.help —
ExpertSystemToolsPackage1 contains some useful functions for use by the computational agents of Ordinary Differential Equation solvers.

See Also:
o )show ExpertSystemToolsPackage1

---

ExpertSystemToolsPackage1 (ESTOOLS1)

Exports:

neglist

--- package ESTOOLS1 ExpertSystemToolsPackage1 ---

)abbrev package ESTOOLS1 ExpertSystemToolsPackage1
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: February 1995
++ Description:
++ \texttt{\textbackslash axiom}({ExpertSystemToolsPackage1}) contains some useful functions for use
++ by the computational agents of Ordinary Differential Equation solvers.

ExpertSystemToolsPackage1(R1:OR): E == I where
OR ==> OrderedRing
E ==> with
   neglist:List R1 -> List R1
   ++ neglist(l) returns only the negative elements of the list \texttt{l}
I ==> add
neglist(l:List R1):List R1 == [u for u in l | negative?(u)$R1]

---

ESTOOLS1.dotabb

"ESTOOLS1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS1"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ESTOOLS1" -> "OAGROUP"

package ESTOOLS2 ExpertSystemToolsPackage2

---

ExpertSystemToolsPackage2.input

)set break resume
)sys rm -f ExpertSystemToolsPackage2.output
)spool ExpertSystemToolsPackage2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpertSystemToolsPackage2
--E 1

)spool
)lisp (bye)

---

ExpertSystemToolsPackage2.help

=================================================
ExpertSystemToolsPackage2 examples
=================================================

ExpertSystemToolsPackage2 contains some useful functions for use
by the computational agents of Ordinary Differential Equation solvers.

See Also:
  o )show ExpertSystemToolsPackage2
ExpertSystemToolsPackage2 (ESTOOLS2)

Exports:

map

package ESTOOLS2 ExpertSystemToolsPackage2

++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: July 1996
++ Description:
++ \texttt{\texttt{\texttt{Axim}}Estoools2} contains some useful functions for use
++ by the computational agents of Ordinary Differential Equation solvers.

ExpertSystemToolsPackage2(R1:R,R2:R): E == I where
R == Ring
E == with
  map:(R1->R2, Matrix R1) -> Matrix R2
  ++ map(f,m) applies a mapping f:R1 -> R2 onto a matrix
  ++ \texttt{spad(m)} in R1 returning a matrix in R2
I == add
  map(f:R1->R2,m:Matrix R1):Matrix R2 ==
  matrix([[f u for u in v] for v in listOfLists(m)$(Matrix R1)]$(Matrix R2))
package EXPR2 ExpressionFunctions2

--- ExpressionFunctions2.input ---

)set break resume
)sys rm -f ExpressionFunctions2.output
)spool ExpressionFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionFunctions2
--E 1

)spool
)lisp (bye)

---

--- ExpressionFunctions2.help ---

====================================================================
ExpressionFunctions2 examples
====================================================================

Lifting of maps to Expressions.

See Also:
o )show ExpressionFunctions2

---
ExpressionFunctions2 (EXPR2)

Exports:
map

— package EXPR2 ExpressionFunctions2 —

)abbrev package EXPR2 ExpressionFunctions2
++ Lifting of maps to Expressions
++ Author: Manuel Bronstein
++ Date Created: 16 Jan 1989
++ Date Last Updated: 22 Jan 1990
++ Description:
++ Lifting of maps to Expressions.

ExpressionFunctions2(R:OrderedSet, S:OrderedSet):
Exports == Implementation where
K ==> Kernel R
F2 ==> FunctionSpaceFunctions2(R, Expression R, S, Expression S)
E2 ==> ExpressionSpaceFunctions2(Expression R, Expression S)
Exports ==> with
map: (R -> S, Expression R) -> Expression S
++ map(f, e) applies f to all the constants appearing in e.

Implementation == add
if S has Ring and R has Ring then
  map(f, r) == map(f, r)$F2
else
  map(f, r) == map(x1 +-> map(f, x1), retract r)$E2

———

— EXPR2.dotabb ——
package EXPRSOL ExpressionSolve

Bugs

seriesSolve(sin f x / cos x, f, x, [1])$EXPRSOL(INT, EXPR INT, UFPS EXPR INT, UFPS SUPEXPR EXPR INT)

returns

(((0 . 1) 0 . 1) NonNullStream #<compiled-function |STREAM;generate;M$;62!0|> . UNPRINTABLE)

but

U ==> UFPS SUPEXPR EXPR INT
seriesSolve(s +-> sin s *((cos monomial(1,1)$U)**-1)$U, f, x, [0])$EXPRSOL(INT, EXPR INT, UFPS SUPEXPR EXPR INT)

works. This is probably due to missing “/” in UFPS.

I'd really like to be able to specify a function that works for all domains in a category. For example, x +-> y(x)^2 + sin x + x should work for EXPR INT as well as for UTS INT, both being domains having TranscendentalFunctionCategory.

— ExpressionSolve.input —

)set break resume
)sys rm -f ExpressionSolve.output
)spool ExpressionSolve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSolve
--E 1

)spool
)lisp (bye)
ExpressionSolve (EXPRSOL)

Exports:
replaceDiffs  seriesSolve

— package EXPRSOL ExpressionSolve —

EXPRSOL

UTSSOL

)abbrev package EXPRSOL ExpressionSolve
++ Description:
++ This package has no description
ExpressionSolve(R, F, UTSF, UTSSUPF):Exports==Implementation where
  R:Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
  F:FunctionSpace R
  UTSF:UnivariateTaylorSeriesCategory F
  SUP ==> SparseUnivariatePolynomialExpressions
  UTSSUPF:UnivariateTaylorSeriesCategory SUP F
  OP  ==> BasicOperator
SY ==> Symbol
NNI ==> NonNegativeInteger
MKF ==> MakeBinaryCompiledFunction(F, UTSSUPF, UTSSUPF, UTSSUPF)

Exports == with

    seriesSolve: (F, OP, SY, List F) -> UTSF
    replaceDiffs: (F, OP, Symbol) -> F

Implementation == add

\getchunk{implementation: EXPRSOL ExpressionSolve}

The general method is to transform the given expression into a form which can then be compiled. There is currently no other way in Axiom to transform an expression into a function.

We need to replace the differentiation operator by the corresponding function in the power series category, and make composition explicit. Furthermore, we need to replace the variable by the corresponding variable in the power series. It turns out that the compiler doesn’t find the right definition of monomial(1,1). Thus we introduce it as a second argument. In fact, maybe that’s even cleaner. Also, we need to tell the compiler that kernels that are independent of the main variable should be coerced to elements of the coefficient ring, since it will complain otherwise.

I cannot find an example for this behaviour right now. However, if I do use the coerce, the following fails:

\texttt{seriesSolve(h \times -1-\times h \times h(q \times x), h, x, [1])}

--- package EXPRSOL ExpressionSolve ---

\getchunk{implementation: EXPRSOL ExpressionSolve}

opelt := operator("elt"::Symbol)$OP
opdiff := operator("D"::Symbol)$OP
opcoerce := operator("coerce"::Symbol)$OP

-- replaceDiffs: (F, OP, Symbol) -> F
replaceDiffs (expr, op, sy) ==
    lk := kernels expr
    for k in lk repeat
        if freeOf?(coerce k, sy) then
            expr := subst(expr, [k], [opcoerce [coerce k]])
        if is?(k, op) then
            arg := first argument k

if is?(k, op) then
    arg := first argument k
if arg = sy::F
then expr := subst(expr, [k], [(name op)::F])
else expr := subst(expr, [k], [opelt [(name op)::F,
replaceDiffs(arg, op, sy)]])

-- => "iterate"

if is?(k, %diff) then
  args := argument k
differentiand :=
  replaceDiffs(subst(args.1, args.2 = args.3), op, sy)
expr := subst(expr, [k], [opdiff differentiand])

-- => "iterate"

seriesSolve(expr, op, sy, l) ==
ex := replaceDiffs(expr, op, sy)
f := compiledFunction(ex, name op, sy)$MKF
seriesSolve(x+->f(x, monomial(1,1)$UTSSUPF), l)_
$TaylorSolve(F, UTSF, UTSSUPF)

-- EXPRSOL.dotabb --
"EXPRSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRSOL"]
"UTSSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSSOL"]
"EXPRSOL" -> "UTSSOL"

package ES1 ExpressionSpaceFunctions1

— ExpressionSpaceFunctions1.input —

)set break resume
)sys rm -f ExpressionSpaceFunctions1.output
)spool ExpressionSpaceFunctions1.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show ExpressionSpaceFunctions1
ExpressionSpaceFunctions1 (ES1)

Exports:
map

--- package ES1 ExpressionSpaceFunctions1 ---

)abbrev package ES1 ExpressionSpaceFunctions1
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a map from any expression space into any object
++ to be lifted to a kernel over the expression set, using a given
++ property of the operator of the kernel.
-- should not be exposed

ExpressionSpaceFunctions1(F:ExpressionSpace, S:Type): with
  map: (F -> S, String, Kernel F) -> S
  ++ map(f, p, k) uses the property p of the operator
  ++ of k, in order to lift f and apply it to k.

== add
  -- prop contains an evaluation function List F -> S
  map(F2S, prop, k) ==
  args := [F2S x for x in argument k]$List(S)
  (p := property(operator k, prop)) case None =>
  ((p::None) pretend (List S -> S)) args
  error "Operator does not have required property"

— ES1.dotabb —

"ES1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES1"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"ES1" -> "EVALAB"

— package ES2 ExpressionSpaceFunctions2 —

package ES2 ExpressionSpaceFunctions2

— ExpressionSpaceFunctions2.input —

)set break resume
)sys rm -f ExpressionSpaceFunctions2.output
)spool ExpressionSpaceFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSpaceFunctions2
--E 1
ExpressionSpaceFunctions2 (ES2)

Exports:
map

See Also:
o )show ExpressionSpaceFunctions2

This package allows a mapping E -> F to be lifted to a kernel over E; This lifting can fail if the operator of the kernel cannot be applied in F; Do not use this package with E = F, since this may drop some properties of the operators.
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a mapping \( E \to F \) to be lifted to a kernel over \( E \);
++ This lifting can fail if the operator of the kernel cannot be applied
++ in \( F \); Do not use this package with \( E = F \), since this may
++ drop some properties of the operators.

ExpressionSpaceFunctions2(E:ExpressionSpace, F:ExpressionSpace): with
  map: (E -> F, Kernel E) -> F
  ++ map(f, k) returns \( \text{spad}(g = \text{op}(f(a_1),\ldots,f(a_n))) \) where
  ++ \( \text{spad}(k = \text{op}(a_1,\ldots,a_n)) \).
  == add
  map(f, k) ==
  (operator(operator k)$F) [f x for x in argument k]$List(F)

———

— ES2.dotabb —

"ES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES2"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"ES2" -> "EVALAB"

———

package EXPRODE ExpressionSpaceODESolver

— ExpressionSpaceODESolver.input —

)set break resume
)sys rm -f ExpressionSpaceODESolver.output
)spool ExpressionSpaceODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSpaceODESolver
--E 1

)spool
)lisp (bye)

———
expressionSpaceODESolver.help

ExpressionSpaceODESolver examples

Taylor series solutions of explicit ODE's;

See Also:
o )show ExpressionSpaceODESolver

ExpressionSpaceODESolver (EXPRODE)

Exports:
  seriesSolve

package EXPRODE ExpressionSpaceODESolver
P ==> SparseMultivariatePolynomial(R, K)
OP ==> BasicOperator
SY ==> Symbol
UTS ==> UnivariateTaylorSeries(F, x, center)
MKF ==> MakeUnaryCompiledFunction(F, UTS, UTS)
MKL ==> MakeUnaryCompiledFunction(F, List UTS, UTS)
A1 ==> AnyFunctions1(UTS)
AL1 ==> AnyFunctions1(List UTS)
EQ ==> Equation F
ODE ==> UnivariateTaylorSeriesODESolver(F, UTS)

Exports ==> with
seriesSolve: (EQ, OP, EQ, EQ) -> Any
  ++ seriesSolve(eq,y,x=a, y a = b) returns a Taylor series solution
  ++ of eq around x = a with initial condition \spad{y(a) = b}.
  ++ Note that eq must be of the form
  ++ \spad{y'(x) + g(x, y x) h(x, y x)}.
seriesSolve: (EQ, OP, EQ, List F) -> Any
  ++ seriesSolve(eq,y,x=a,[b0,...,b(n-1)]) returns a Taylor series
  ++ solution of eq around \spad{x = a} with initial conditions
  ++ \spad{y(a) = b0}, \spad{y'(a) = b1},
  ++ \spad{y''(a) = b2}, ..., \spad{y(n-1)(a) = b(n-1)}
  ++ eq must be of the form
  ++ \spad{y'(x) + g(x,y x, y'(x),..., y(n-1)(x)) h(x,y x, y'(x),..., y(n-1)(x))}.
seriesSolve: (List EQ, List OP, EQ, List EQ) -> Any
  ++ seriesSolve([eq1,...,eqn],[y1,...,yn],x = a,[y1 a = b1,...,yn a = bn])
  ++ returns a taylor series solution of \spad{[eq1,...,eqn]} around
  ++ \spad{x = a} with initial conditions \spad{yi(a) = bi}.
  ++ Note that eqi must be of the form
  ++ \spad{y'(x) + g(x,y1 x, y2 x,..., yn x) h(x,y1 x, y2 x,..., yn x)}.
seriesSolve: (List F, List OP, EQ, List F) -> Any
  ++ seriesSolve([eq1=0,...,eqn=0], [y1,...,yn], x=a, [b1,...,bn])
  ++ is equivalent to
  ++ \spad{seriesSolve([eq1,...,eqn], [y1,...,yn], x = a,}
  ++ [y1 a = b1,..., yn a = bn]}.
seriesSolve: (EQ, OP, EQ, F) -> Any
  ++ seriesSolve(eq,y,x=a, b) is equivalent to
  ++ \spad{seriesSolve(eq, y, x=a, y a = b)}. 
seriesSolve: (F, OP, EQ, F) -> Any
++ seriesSolve(eq, y, x = a, b) is equivalent to
++ \spad{seriesSolve(eq = 0, y, x = a, y a = b)}.
seriesSolve: (F, OP, EQ, EQ) -> Any
++ seriesSolve(eq, y, x = a, y a = b) is equivalent to
++ \spad{seriesSolve(eq = 0, y, x = a, y a = b)}.
seriesSolve: (F, OP, EQ, List F) -> Any
++ seriesSolve(eq, y, x = a, [b0,...,bn]) is equivalent to
++ \spad{seriesSolve(eq = 0, y, x = a, [b0,...,b(n-1)])}.

Implementation ==> add

checkCompat: (OP, EQ, EQ) -> F
checkOrder1: (F, OP, K, SY, F) -> F
checkOrderN: (F, OP, K, SY, F, NonNegativeInteger) -> F
checkSystem: (F, List K, List F) -> F
div2exquo : F -> F
smp2exquo : P -> F
k2exquo : K -> F
diffRhs : (F, F) -> F
diffRhsK : (K, F) -> F
findCompat : (F, List EQ) -> F
findEq : (K, SY, List F) -> F
localInteger: F -> F

opelt := operator("elt"::Symbol)$OP
--opex := operator("exquo"::Symbol)$OP
opex := operator("fixedPointExquo"::Symbol)$OP
opint := operator("integer"::Symbol)$OP

Rint? := R has IntegerNumberSystem

localInteger n == (Rint? => n; opint n)
diffRhs(f, g) == diffRhsK(retract(f)@K, g)

k2exquo k ==
  is?(op := operator k, "%diff"::Symbol) =>
  error "Improper differential equation"
  kernel(op, [div2exquo f for f in argument k]$List(F))

smp2exquo p ==
  map(k2exquo,x->x::F,p)_
  $PolynomialCategoryLifting(IndexedExponents K,K,R,P,F)

div2exquo f ==
  -- one?(d := denom f) => f
  ((d := denom f) = 1) => f
  opex(smp2exquo numer f, smp2exquo d)

  -- if g is of the form a * k + b, then return -b/a
diffRhsK(k, g) ==
h := univariate(g, k)
(degree(numer h) <= 1 and ground? denom h =>
  - coefficient(numer h, 0) / coefficient(numer h, 1)
) error "Improper differential equation"

checkCompat(y, eqx, eqy) ==
  lhs(eqy) =$F y(rhs eqx) => rhs eqy
  error "Improper initial value"

findCompat(yx, 1) ==
  for eq in 1 repeat
    yx =$F lhs eq => return rhs eq
  error "Improper initial value"

findEq(k, x, sys) ==
  k := retract(differentiate(k::F, x))@K
  for eq in sys repeat
    member?(k, kernels eq) => return eq
  error "Improper differential equation"

checkOrder1(diffeq, y, yx, x, sy) ==
  div2exquo subst(diffRhs(differentiate(yx::F,x),diffeq),[yx],[sy])

checkOrderN(diffeq, y, yx, x, sy, n) ==
  zero? n => error "No initial value(s) given"
  m := (minIndex(l := [retract(f := yx::F)@K]$List(K)))::F
  lv := [opelt(sy, localInteger m)]$List(F)
  for i in 2..n repeat
    l := concat(retract(f := differentiate(f, x))@K, l)
    lv := concat(opelt(sy, localInteger(m := m + 1)), lv)
    div2exquo subst(diffRhs(differentiate(f, x), diffeq), l, lv)

checkSystem(diffeq, yx, lv) ==
  for k in kernels diffeq repeat
    is?(k, "%diff"::SY) =>
      return div2exquo subst(diffRhsK(k, diffeq), yx, lv)
    0

seriesSolve(l:List EQ, y:List OP, eqx:EQ, eqy:List EQ) ==
  seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, eqy)

seriesSolve(l:List EQ, y:List OP, eqx:EQ, y0:List F) ==
  seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, y0)

seriesSolve(l:List F, ly:List OP, eqx:EQ, eqy:List EQ) ==
  seriesSolve(l, ly, eqx,
              [findCompat(y rhs eqx, eqy) for y in ly]$List(F))

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, eqy:EQ) ==
  seriesSolve(lhs diffeq - rhs diffeq, y, eqx, eqy)
seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:F) ==
seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:List F) ==
seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

seriesSolve(diffeq:F, y:OP, eqx:EQ, eqy:EQ) ==
seriesSolve(diffeq, y, eqx, checkCompat(y, eqx, eqy))

seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:F) ==
  x := symbolIfCan(retract(lhs eqx)@K)::SY
  sy := name y
  yx := retract(y lhs eqx)@K
  f := checkOrder1(diffeq, y, yx, x, sy::F)
  center := rhs eqx
  coerce(ode1(compiledFunction(f, sy)$MKF, y0)$ODE)$A1

seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:List F) ==
  x := symbolIfCan(retract(lhs eqx)@K)::SY
  sy := new()$SY
  yx := retract(y lhs eqx)@K
  f := checkOrderN(diffeq, y, yx, x, sy::F, #y0)
  center := rhs eqx
  coerce(ode(compiledFunction(f, sy)$MKL, y0)$ODE)$A1

seriesSolve(sys:List F, ly:List OP, eqx:EQ, l0:List F) ==
  x := symbolIfCan(kx := retract(lhs eqx)@K)::SY
  fsy := (sy := new()$SY)::F
  m := (minIndex(l0) - 1)::F
  yx := concat(kx, [retract(y lhs eqx)@K for y in ly]$List(K))
  lelt := [opelt(fsy, localInteger(m := m+1)) for k in yx]$List(F)
  sys := [findEq(k, x, sys) for k in rest yx]
  l := [checkSystem(eq, yx, lelt) for eq in sys]$List(F)
  center := rhs eqx
  coerce(mpsode(l0,[compiledFunction(f,sy)$MKL for f in l])$ODE)$A1

— EXPRODE.dotabb —

"EXPRODE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRODE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EXPRODE" -> "FS"
package OMEXPR ExpressionToOpenMath

--- ExpressionToOpenMath.input ---

)set break resume
)sys rm -f ExpressionToOpenMath.output
)spool ExpressionToOpenMath.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionToOpenMath
--E 1

)spool
)lisp (bye)

---

--- ExpressionToOpenMath.help ---

====================================================================
ExpressionToOpenMath examples
====================================================================

ExpressionToOpenMath provides support for converting objects of type
Expression into OpenMath.

See Also:
o )show ExpressionToOpenMath

---
ExpressionToOpenMath (OMEXPR)

Exports:
OMwrite

— package OMEXPR ExpressionToOpenMath —

)abbrev package OMEXPR ExpressionToOpenMath
++ Author: Mike Dewar & Vilya Harvey
++ Description:
++ \spadtype{ExpressionToOpenMath} provides support for
++ converting objects of type \spadtype{Expression} into OpenMath.

ExpressionToOpenMath(R: Join(OpenMath, OrderedSet, Ring)): with
  OMwrite : Expression R -> String
  OMwrite : (Expression R, Boolean) -> String
  OMwrite : (OpenMathDevice, Expression R) -> Void
  OMwrite : (OpenMathDevice, Expression R, Boolean) -> Void
== add
  import Expression R
  SymInfo ==> Record(cd:String, name:String)
  import SymInfo
  import Record(key: Symbol, entry: SymInfo)
  import AssociationList(Symbol, SymInfo)
  import OMENC

-------------------------
-- Local translation tables.
-------------------------

nullaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_,
                       [pi, ["nums1", "pi"]]]

unaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_,
                      [exp, ["transc1", "exp"]],_
                      [log, ["transc1", "ln"]],_
                      [sin, ["transc1", "sin"]],_
[\cos, \text{["transc1", "cos"]}],
[tan, \text{["transc1", "tan"]}],
[cot, \text{["transc1", "cot"]}],
[sec, \text{["transc1", "sec"]}],
[csc, \text{["transc1", "csc"]}],
[asin, \text{["transc1", "arcsin"]}],
[acos, \text{["transc1", "arccos"]}],
[atan, \text{["transc1", "arctan"]}],
[acot, \text{["transc1", "arccot"]}],
[asec, \text{["transc1", "arcsec"]}],
[acsc, \text{["transc1", "arccsc"]}],
[sinh, \text{["transc1", "sinh"]}],
[cosh, \text{["transc1", "cosh"]}],
[tanh, \text{["transc1", "tanh"]}],
[coth, \text{["transc1", "coth"]}],
[sech, \text{["transc1", "sech"]}],
[cosech, \text{["transc1", "cosech"]}],
[asinh, \text{["transc1", "arcsinh"]}],
[acosh, \text{["transc1", "arccosh"]}],
[atanh, \text{["transc1", "arctanh"]}],
[acoth, \text{["transc1", "arccoth"]}],
[asech, \text{["transc1", "arcsech"]}],
[acsch, \text{["transc1", "arccsch"]}],
[factorial, \text{["integer1", "factorial"]}],
[abs, \text{["arith1", "abs"]}] ]

-- Still need the following unary functions:
-- digamma
-- Gamma
-- airyAi
-- airyBi
-- erf
-- Ei
-- Si
-- Ci
-- li
-- dilog

-- Still need the following binary functions:
-- Gamma(a, x)
-- Beta(x, y)
-- polygamma(k, x)
-- besselJ(v, x)
-- besselY(v, x)
-- besselI(v, x)
-- besselK(v, x)
-- permutation(n, m)
-- summation(x:%, n:Symbol) : as opposed to "definite" sum
-- product(x:%, n:Symbol) : ditto
--- Forward declarations.

outputOMExpr : (OpenMathDevice, Expression R) -> Void

--- Local helper functions

outputOMArith1(dev: OpenMathDevice, sym: String, args: List Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "arith1", sym)
for arg in args repeat
  OMwrite(dev, arg, false)
OMputEndApp(dev)

outputOMLambda(dev: OpenMathDevice, ex: Expression R, var: Expression R): Void ==
OMputBind(dev)
OMputSymbol(dev, "fns1", "lambda")
OMwrite(dev, var, false)
OMputEndBVar(dev)
OMwrite(dev, ex, false)
OMputEndBind(dev)

outputOMInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "interval1", "interval")
OMwrite(dev, lo, false)
OMwrite(dev, hi, false)
OMputEndApp(dev)

outputOMIntInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "interval1", "integer__interval")
OMwrite(dev, lo, false)
OMwrite(dev, hi, false)
OMputEndApp(dev)

outputOMBinomial(dev: OpenMathDevice, args: List Expression R): Void ==
not #args=2 => error "Wrong number of arguments to binomial"
OMputApp(dev)
OMputSymbol(dev, "combinat1", "binomial")
for arg in args repeat
  OMwrite(dev, arg, false)
OMputEndApp(dev)

outputOMPower(dev: OpenMathDevice, args: List Expression R): Void ==
not #args=2 => error "Wrong number of arguments to power"
outputOMArith(dev, "power", args)

outputOMDefsum(dev: OpenMathDevice, args: List Expression R): Void ==
    #args ^= 5 => error "Unexpected number of arguments to a defsum"
    OMputApp(dev)
    OMputSymbol(dev, "arith1", "sum")
    outputOMIntInterval(dev, args.4, args.5)
    outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
    OMputEndApp(dev)

outputOMDefprod(dev: OpenMathDevice, args: List Expression R): Void ==
    #args ^= 5 => error "Unexpected number of arguments to a defprod"
    OMputApp(dev)
    OMputSymbol(dev, "arith1", "product")
    outputOMIntInterval(dev, args.4, args.5)
    outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
    OMputEndApp(dev)

outputOMDefint(dev: OpenMathDevice, args: List Expression R): Void ==
    #args ^= 5 => error "Unexpected number of arguments to a defint"
    OMputApp(dev)
    OMputSymbol(dev, "calculus1", "defint")
    outputOMInterval(dev, args.4, args.5)
    outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
    OMputEndApp(dev)

outputOMInt(dev: OpenMathDevice, args: List Expression R): Void ==
    #args ^= 3 => error "Unexpected number of arguments to a defint"
    OMputApp(dev)
    OMputSymbol(dev, "calculus1", "int")
    outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
    OMputEndApp(dev)

outputOMFunction(dev: OpenMathDevice, op: Symbol, args: List Expression R): Void ==
    nargs := #args
    zero? nargs =>
        omOp: Union(SymInfo, "failed") := search(op, nullaryFunctionAList)
        omOp case "failed" =>
            error concat ["No OpenMath definition for nullary function ", coercer op]
            OMputSymbol(dev, omOp.cd, omOp.name)
    one? nargs =>
        (nargs = 1) =>
            omOp: Union(SymInfo, "failed") := search(op, unaryFunctionAList)
            omOp case "failed" =>
                error concat ["No OpenMath definition for unary function ", coercer op]
                OMputApp(dev)
            OMputSymbol(dev, omOp.cd, omOp.name)
            for arg in args repeat
                OMwrite(dev, arg, false)
            OMputEndApp(dev)
-- Most of the binary operators cannot be handled trivially like the
-- unary ones since they have bound variables of one kind or another.
-- The special functions should be straightforward, but we don't have
-- a CD for them yet :-)

op = %defint => outputOMDefint(dev, args)
op = integral => outputOMInt(dev, args)
op = %defsum => outputOMDefsum(dev, args)
op = %defprod => outputOMDefprod(dev, args)
op = %power => outputOMPower(dev, args)
op = binomial => outputOMBinomial(dev, args)
error concat ["No OpenMath definition for function ", string op]

outputOMExpr(dev: OpenMathDevice, ex: Expression R): Void ==
ground? ex => OMwrite(dev, ground ex, false)
not((v := retractIfCan(ex)@Union(Symbol,"failed")) case "failed") =>
  OMputVariable(dev, v)
not((w := isPlus ex) case "failed") => outputOMArith1(dev, "plus", w)
not((w := isTimes ex) case "failed") => outputOMArith1(dev, "times", w)
--not((y := isMult ex) case "failed") =>
--  outputOMArith("times", [OMwrite(y.coef)$Integer,
--    OMwrite(coerce y.var)])
-- At the time of writing we don't need both isExpt and isPower
-- here but they may be relevent when we integrate this stuff into
-- the main Expression code. Note that if we don't check that
-- the exponent is non-trivial we get thrown into an infinite recursion.
-- not (((x := isExpt ex) case "failed") or one? x.exponent) =>
-- not (((x := isExpt ex) case "failed") or (x.exponent = 1)) =>
  not((s := symbolIfCan(x.var)@Union(Symbol,"failed")) case "failed") =>
    --outputOMPower(dev, [s::Expression(R), (x.exponent)::Expression(R)])
    OMputApp(dev)
    OMputSymbol(dev, "arith1", "power")
    OMputVariable(dev, s)
    OMputInteger(dev, x.exponent)
    OMputEndApp(dev)
-- TODO: add error handling code here...
-- not (((z := isPower ex) case "failed") or one? z.exponent) =>
-- not (((z := isPower ex) case "failed") or (z.exponent = 1)) =>
  outputOMPower(dev, [ z.val, z.exponent::Expression R ])
  OMputApp(dev)
  OMputSymbol(dev, "arith1", "power")
  --outputOMExpr(dev, z.val)
  OMputInteger(dev, z.exponent)
  OMputEndApp(dev)
-- Must only be one top-level Kernel by this point
k : Kernel Expression R := first kernels ex
outputOMFunction(dev, name operator k, argument k)

----------
-- Exports
---

OMwrite(ex: Expression R): String ==

  s := ""
  sp := OM_STRINGTOSTRINGPTR(s)
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
  OMputObject(dev)
  outputOMExpr(dev, ex)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_STRINGPTRTOSTRING(sp)
  s

OMwrite(ex: Expression R, wholeObj: Boolean): String ==

  s := ""
  sp := OM_STRINGTOSTRINGPTR(s)
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
  if wholeObj then
    OMputObject(dev)
    outputOMExpr(dev, ex)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_STRINGPTRTOSTRING(sp)
  s

OMwrite(dev: OpenMathDevice, ex: Expression R): Void ==

  OMputObject(dev)
  outputOMExpr(dev, ex)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, ex: Expression R, wholeObj: Boolean): Void ==

  if wholeObj then
    OMputObject(dev)
    outputOMExpr(dev, ex)
  if wholeObj then
    OMputEndObject(dev)

---

— OMEXPR.dotabb —

"OMEXPR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMEXPR"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"OMEXPR" -> "FS"
package EXPR2UPS ExpressionToUnivariatePowerSeries

--- ExpressionToUnivariatePowerSeries.input ---

)set break resume
)sys rm -f ExpressionToUnivariatePowerSeries.output
)spool ExpressionToUnivariatePowerSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionToUnivariatePowerSeries
--E 1

)spool
)lisp (bye)

--- ExpressionToUnivariatePowerSeries.help ---

====================================================================
ExpressionToUnivariatePowerSeries examples
====================================================================

This package provides functions to convert functional expressions to power series.

See Also:
o )show ExpressionToUnivariatePowerSeries

---
ExpressionToUnivariatePowerSeries (EXPR2UPS)

Exports:
laurent puiseux series taylor

— package EXPR2UPS ExpressionToUnivariatePowerSeries —

)abbrev package EXPR2UPS ExpressionToUnivariatePowerSeries
+ Author: Clifton J. Williamson
++ Date Created: 9 May 1989
++ Date Last Updated: 20 September 1993
++ Description:
++ This package provides functions to convert functional expressions
++ to power series.

ExpressionToUnivariatePowerSeries(R,FE): Exports == Implementation where
  R : Join(GcdDomain,OrderedSet,RetractableTo Integer,_,
          LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_,
            FunctionSpace R)

EQ ==> Equation
I ==> Integer
NNI ==> NonNegativeInteger
RN ==> Fraction Integer
SY ==> Symbol
UTS ==> UnivariateTaylorSeries
ULS ==> UnivariateLaurentSeries
UPXS ==> UnivariatePuiseuxSeries
GSER ==> GeneralUnivariatePowerSeries
EFULS ==> ElementaryFunctionsUnivariateLaurentSeries
EFUPXS ==> ElementaryFunctionsUnivariatePuiseuxSeries
FS2UPS ==> FunctionSpaceToUnivariatePowerSeries
Prob ==> Record(func:String,prob:String)
ANY1 ==> AnyFunctions1

Exports ==> with
taylor: SY -> Any
++ \spad{taylor(x)} returns x viewed as a Taylor series.
taylor: FE -> Any
++ \spad{taylor(f)} returns a Taylor expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
taylor: (FE,NNI) -> Any
++ \spad{taylor(f,n)} returns a Taylor expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
taylor: (FE,EQ FE) -> Any
++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
++ in powers of \spad{(x - a)}.
taylor: (FE,EQ FE,NNI) -> Any
++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
++ in powers of \spad{(x - a)}; terms will be computed up to order
++ at least n.

laurent: SY -> Any
++ \spad{laurent(x)} returns x viewed as a Laurent series.
laurent: FE -> Any
++ \spad{laurent(f)} returns a Laurent expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
laurent: (FE,I) -> Any
++ \spad{laurent(f,n)} returns a Laurent expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
laurent: (FE,EQ FE) -> Any
++ \spad{laurent(f,x = a)} expands the expression f as a Laurent series
++ in powers of \spad{(x - a)}.
laurent: (FE,EQ FE,I) -> Any
++ \spad{laurent(f,x = a,n)} expands the expression f as a Laurent
++ series in powers of \spad{(x - a)}; terms will be computed up to order
++ at least n.
puiseux: SY -> Any
++ \spad{puiseux(x)} returns x viewed as a Puiseux series.
puiseux: FE -> Any
++ \spad{puiseux(f)} returns a Puiseux expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
puiseux: (FE,RN) -> Any
++ \spad{puiseux(f,n)} returns a Puiseux expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
puiseux: (FE,EQ FE) -> Any
++ \spad{puiseux(f,x = a)} expands the expression f as a Puiseux series
++ in powers of \spad{(x - a)}.

puiseux: \(\text{FE, EQ, FE, RN} \rightarrow \text{Any}\)
++ \spad{puiseux(f, x = a, n)} expands the expression \(f\) as a Puiseux
++ series in powers of \spad{(x - a)}; terms will be computed up to order
++ at least \(n\).

series: \(\text{SY} \rightarrow \text{Any}\)
++ \spad{series(x)} returns \(x\) viewed as a series.

series: \(\text{FE} \rightarrow \text{Any}\)
++ \spad{series(f)} returns a series expansion of the expression \(f\).
++ Note that \(f\) should have only one variable; the series will be
++ expanded in powers of that variable.

series: \(\text{FE, RN} \rightarrow \text{Any}\)
++ \spad{series(f, n)} returns a series expansion of the expression \(f\).
++ Note that \(f\) should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least \(n\).

series: \(\text{FE, EQ, FE} \rightarrow \text{Any}\)
++ \spad{series(f, x = a)} expands the expression \(f\) as a series
++ in powers of \(x - a\).

series: \(\text{FE, EQ, FE, RN} \rightarrow \text{Any}\)
++ \spad{series(f, x = a, n)} expands the expression \(f\) as a series
++ in powers of \(x - a\); terms will be computed up to order
++ at least \(n\).

Implementation ==> add
performSubstitution: \(\text{FE, SY, FE} \rightarrow \text{FE}\)
performSubstitution(fcn, x, a) ==
  if a = 0 then fcn
  xFE := x :: \text{FE}
  eval(fcn, xFE = xFE + a)

iTaylor: \(\text{FE, SY, FE} \rightarrow \text{Any}\)
iTaylor(fcn, x, a) ==
  pack := \text{FSZUPS}(R, FE, I, ULS(FE, x, a),
                      EFULS(FE, UTS(FE, x, a), ULS(FE, x, a)), x)
  ans := exprToUPS(fcn, false, "just do it")$pack
  ans case %problem =>
    ans.%problem.prob = "essential singularity" =>
      error "No Taylor expansion: essential singularity"
    ans.%problem.func = "log" =>
      error "No Taylor expansion: logarithmic singularity"
    ans.%problem.func = "nth root" =>
      error "No Taylor expansion: fractional powers in expansion"
    error "No Taylor expansion"
  uls := ans.%series
  (uts := taylorIfCan uls) case "failed" =>
    error "No Taylor expansion: pole"
  any1 := ANY1(UTS(FE, x, a))
  coerce(uts :: UTS(FE, x, a))$any1
CHAPTER 6. CHAPTER E

\begin{verbatim}
taylor(x:SY) ==
  uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
  coerce(monomial(1,1)$uts)$any1

  taylor(fcn:FE) ==
    null(vars := variables fcn) =>
      error "taylor: expression has no variables"
    not null rest vars =>
      error "taylor: expression has more than one variable"
    taylor(fcn,(first(vars) :: FE) = 0)

  taylor(fcn:FE,n:NNI) ==
    null(vars := variables fcn) =>
      error "taylor: expression has no variables"
    not null rest vars =>
      error "taylor: expression has more than one variable"
    x := first vars
    uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
    series := retract(taylor(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1

  taylor(fcn,eq:EQ FE) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    iTaylor(performSubstitution(fcn,x,a),x,a)

  taylor(fcn,eq,n) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    any1 := ANY1(UTS(FE,x,a))
    series := retract(iTaylor(performSubstitution(fcn,x,a),x,a))$any1
    coerce(extend(series,n))$any1

iLaurent: (FE,SY,FE) -> Any
iLaurent(fcn,x,a) ==
  pack := FS2UPS(R,FE,I,ULS(FE,x,a),-
    EFULS(FE,UTS(FE,x,a),ULS(FE,x,a)),x)
  ans := exprToUPS(fcn,false,"just do it")$pack
  ans case %problem =>
    ans.%problem.prob = "essential singularity" =>
      error "No Laurent expansion: essential singularity"
    ans.%problem.func = "log" =>
      error "No Laurent expansion: logarithmic singularity"
    ans.%problem.func = "nth root" =>
      error "No Laurent expansion: fractional powers in expansion"
    error "No Laurent expansion"
  any1 := ANY1(ULS(FE,x,a))
\end{verbatim}
coerce(ans.%series)$any1

laurent(x:SY) ==
  uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
  coerce(monomial(1,1)$uls)$any1

laurent(fcn:FE) ==
  null(vars := variables fcn) =>
    error "laurent: expression has no variables"
  not null rest vars =>
    error "laurent: expression has more than one variable"
  laurent(fcn,(first(vars) :: FE) = 0)

laurent(fcn:FE,n:I) ==
  null(vars := variables fcn) =>
    error "laurent: expression has no variables"
  not null rest vars =>
    error "laurent: expression has more than one variable"
  x := first vars
  uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
  series := retract(laurent(fcn,(x :: FE) = 0))$any1
  coerce(extend(series,n))$any1

laurent(fcn:FE,eq:EQ FE) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  iLaurent(performSubstitution(fcn,x,a),x,a)

laurent(fcn,eq,n) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  any1 := ANY1(ULS(FE,x,a))
  series := retract(iLaurent(performSubstitution(fcn,x,a),x,a))$any1
  coerce(extend(series,n))$any1

iPuiseux: (FE,SY,FE) -> Any
iPuiseux(fcn,x,a) ==
  pack := FS2UPS(R,FE,RN,UPXS(FE,x,a),_
  EFUPXS(FE,ULS(FE,x,a),UPXS(FE,x,a),_  
  EFULS(FE,UTS(FE,x,a),ULS(FE,x,a)))$x)
  ans := exprToUPS(fcn,false,"just do it")$pack
  ans case %problem =>
    ans.%problem.prob = "essential singularity" =>
      error "No Puiseux expansion: essential singularity"
    ans.%problem.func = "log" =>
      error "No Puiseux expansion: logarithmic singularity"
    else ans case %problem =>
      error "No Puiseux expansion"
  any1 := ANY1(UPXS(FE,x,a))
coerce(ans, %series)$any1

puiseux(x: SY) ==
  upxs := UPXS(FE, x, 0$FE); any1 := ANY1(upxs)
  coerce(monomial(1, 1)$upxs)$any1

puiseux(fcn: FE) ==
  null(vars := variables fcn) =>
    error "puiseux: expression has no variables"
  not null rest vars =>
    error "puiseux: expression has more than one variable"
  puiseux(fcn, (first(vars) :: FE) = 0)

puiseux(fcn: FE, n: RN) ==
  null(vars := variables fcn) =>
    error "puiseux: expression has no variables"
  not null rest vars =>
    error "puiseux: expression has more than one variable"
  x := first vars
  upxs := UPXS(FE, x, 0$FE); any1 := ANY1(upxs)
  series := retract(puiseux(fcn, (x :: FE) = 0))$any1
  coerce(extend(series, n))$any1

puiseux(fcn: FE, eq: EQ FE) ==
  (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  iPuiseux(performSubstitution(fcn, x, a), x, a)

puiseux(fcn, eq, n) ==
  (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  any1 := ANY1(UPXS(FE, x, a))
  series := retract(iPuiseux(performSubstitution(fcn, x, a), x, a))$any1
  coerce(extend(series, n))$any1

iSeries: (FE, SY, FE) -> Any
iSeries(fcn, x, a) ==
  pack := FS2UPS(R, FE, RN, UPXS(FE, x, a), _,
    EFUPXS(FE, ULS(FE, x, a), UPXS(FE, x, a), _
    EFULS(FE, UTS(FE, x, a), ULS(FE, x, a))$x))
  ans := exprToUPS(fcn, false, "just do it")$pack
  ans case %problem =>
    ansG := exprToGenUPS(fcn, false, "just do it")$pack
    ansG case %problem =>
      ansG.%problem.prob = "essential singularity" =>
        error "No series expansion: essential singularity"
      error "No series expansion"
      anyone := ANY1(GSER(FE, x, a))
coerce((ansG.%series) :: GSER(FE,x,a))$anyone
any1 := ANY1(UPXS(FE,x,a))
coerce(ans.%series)$any1

series(x:SY) ==
upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
coerce(monomial(1,1)$upxs)$any1

series(fcn:FE) ==
null(vars := variables fcn) =>
  error "series: expression has no variables"
not null rest vars =>
  error "series: expression has more than one variable"
series(fcn,(first(vars) :: FE) = 0)

series(fcn:FE,n:RN) ==
null(vars := variables fcn) =>
  error "series: expression has no variables"
not null rest vars =>
  error "series: expression has more than one variable"
x := first vars
upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
series := retract(series(fcn,(x :: FE) = 0))$any1
coerce(extend(series,n))$any1

series(fcn:FE,eq:EQ FE) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
  error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
iSeries(performSubstitution(fcn,x,a),x,a)

series(fcn,eq,n) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
  error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
any1 := ANY1(UPXS(FE,x,a))
series := retract(iSeries(performSubstitution(fcn,x,a),x,a))$any1
coerce(extend(series,n))$any1
package EXPRTUBE ExpressionTubePlot

— ExpressionTubePlot.input —

)set break resume
)sys rm -f ExpressionTubePlot.output
)spool ExpressionTubePlot.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionTubePlot
--E 1

)spool
)lisp (bye)

— ExpressionTubePlot.help —

===============================================
ExpressionTubePlot examples
===============================================

Package for constructing tubes around 3-dimensional parametric curves.

See Also:
- )show ExpressionTubePlot
ExpressionTubePlot (EXPRTUBE)

Exports:
constantToUnaryFunction   tubePlot

— package EXPRTUBE ExpressionTubePlot —

abbrev package EXPRTUBE ExpressionTubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.

ExpressionTubePlot(): Exports == Implementation where
B ==> Boolean
I ==> Integer
FE ==> Expression Integer
SY ==> Symbol
SF ==> DoubleFloat
L ==> List
S ==> String
SEG ==> Segment
F2F ==> MakeFloatCompiledFunction(FE)
Pt ==> Point SF
PLOT3 ==> Plot3D
TUBE ==> TubePlot Plot3D

Exports == with
constantToUnaryFunction: SF -> (SF -> SF)
++ constantToUnaryFunction(s) is a local function which takes the
++ value of s, which may be a function of a constant, and returns
++ a function which always returns the value \texttt{DoubleFloat} s.
tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF -> SF,I) -> TUBE
++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius r(t) with
++ n points on each circle about the curve \texttt{DoubleFloat} f(t),
++ \texttt{DoubleFloat} g(t), \texttt{DoubleFloat} h(t) for t in \texttt{DoubleFloat} [a,b].
++ The tube is considered to be open.
tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF -> SF,I,S) -> TUBE
  ++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius \( r(t) \)
  ++ with \( n \) points on each circle about the curve \( x = f(t) \),
  ++ \( y = g(t) \), \( z = h(t) \) for \( t \ in [a,b] \).
  ++ If \( s = "closed" \), the tube is ++ considered to be closed; if \( s = "open" \), the tube is considered ++ to be open.

++ The tube is considered to be open.
tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF,I,S) -> TUBE
  ++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius \( r \) with ++ \( n \) points on each circle about the curve \( x = f(t) \),
  ++ \( y = g(t) \), \( z = h(t) \) for \( t \ in [a,b] \).
  ++ The tube is considered to be open.

tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF,I,S) -> TUBE
  ++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius \( r \) with ++ \( n \) points on each circle about the curve \( x = f(t) \),
  ++ \( y = g(t) \), \( z = h(t) \) for \( t \ in [a,b] \).
  ++ If \( s = "closed" \), the tube is ++ considered to be closed; if \( s = "open" \), the tube is considered ++ to be open.

Implementation ==> add
import Plot3D
import F2F
import TubePlotTools

--% variables

getVariable: (FE,FE,FE) -> SY
getVariable(x,y,z) ==
  varList1 := variables x
  varList2 := variables y
  varList3 := variables z
  (not (# varList1 <= 1)) or (not (# varList2 <= 1)) or _
  (not (# varList3 <= 1)) =>
    error "tubePlot: only one variable may be used"
  null varList1 =>
    null varList2 =>
    null varList3 =>
      error "tubePlot: a variable must appear in functions"
      first varList3
    t2 := first varList2
    null varList3 => t2
    not (first varList3 = t2) =>
      error "tubePlot: only one variable may be used"
    t1 := first varList1
    null varList2 =>
    null varList3 => t1
    not (first varList3 = t1) =>
      error "tubePlot: only one variable may be used"
t1
  t2 := first varList2
  null varList3 =>
    not (t1 = t2) =>
      error "tubePlot: only one variable may be used"
  t1
  not (first varList3 = t1) or not (t2 = t1) =>
    error "tubePlot: only one variable may be used"
  t1

--% tubes: variable radius

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF, tRange:SEG SF, radFcn:SF -> SF, n:I, string:S) ==
  -- check value of n
  n < 3 => error "tubePlot: n should be at least 3"
  -- check string
  flag : B :=
    string = "closed" => true
    string = "open" => false
    error "tubePlot: last argument should be open or closed"
  -- check variables
  t := getVariable(x,y,z)
  -- coordinate functions
  xFunc := makeFloatFunction(x,t)
  yFunc := makeFloatFunction(y,t)
  zFunc := makeFloatFunction(z,t)
  -- derivatives of coordinate functions
  xp := differentiate(x,t)
  yp := differentiate(y,t)
  zp := differentiate(z,t)
  -- derivative of arc length
  sp := sqrt(xp ** 2 + yp ** 2 + zp ** 2)
  -- coordinates of unit tangent vector
  Tx := xp/sp; Ty := yp/sp; Tz := zp/sp
  -- derivatives of coordinates of unit tangent vector
  Txp := differentiate(Tx,t)
  Typ := differentiate(Ty,t)
  Tzp := differentiate(Tz,t)
  -- K = curvature = length of curvature vector
  K := sqrt(Txp ** 2 + Typ ** 2 + Tzp ** 2)
  -- coordinates of principal normal vector
  Nx := Txp / K; Ny := Typ / K; Nz := Tzp / K
  -- functions SF->SF giving coordinates of principal normal vector
  NxFunc := makeFloatFunction(Nx,t);
  NyFunc := makeFloatFunction(Ny,t);
  NzFunc := makeFloatFunction(Nz,t);
  -- coordinates of binormal vector
  Bx := Ty * Nz - Tz * Ny
  By := Tz * Nx - Tx * Nz
Bz := Tx * Ny - Ty * Nx
-- functions SF -> SF giving coordinates of binormal vector
BxFunc := makeFloatFunction(Bx,t);
ByFunc := makeFloatFunction(By,t);
BzFunc := makeFloatFunction(Bz,t);
-- create Plot3D
parPlot := plot(xFunc,yFunc,zFunc,colorFcn,tRange)
tvals := first tValues parPlot
curvePts := first listBranches parPlot
cosSin := cosSinInfo n
loopList : L L Pt := nil()
while not null tvals repeat
  -- note that tvals and curvePts have the same number of elements
tval := first tvals; tvals := rest tvals
ctr := first curvePts; curvePts := rest curvePts
pNormList : L SF :=
  [NxFunc tval,NyFunc tval,NzFunc tval,colorFcn tval]
pNorm : Pt := point pNormList
bNormList : L SF :=
  [BxFunc tval,ByFunc tval,BzFunc tval,colorFcn tval]
bNorm : Pt := point bNormList
lps := loopPoints(ctr,pNorm,bNorm,radFcn tval,cosSin)
loopList := cons(lps,loopList)
tube(parPlot,reverse_! loopList,flag)

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,radFcn:SF -> SF,n:I) ==
tubePlot(x,y,z,colorFcn,tRange,radFcn,n,"open")
  --% tubes: constant radius

project: (SF,SF) -> SF
project(x,y) == x
constantToUnaryFunction x == s +-> project(x,s)
tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,rad:SF -> SF,n:I,s:S) ==
tubePlot(x,y,z,colorFcn,tRange,constantToUnaryFunction rad,n,s)
tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,rad:SF,n:I) ==
tubePlot(x,y,z,colorFcn,tRange,rad,n,"open")

— EXPRTUBE.dotabb —
package EXP3D Export3D

--- Export3D.input ---

)set break resume
)sys rm -f Export3D.output
)spool Export3D.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Export3D
--R Export3D is a package constructor
--R Abbreviation for Export3D is EXP3D
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXP3D
--R
--R----------------------------------- Operations -----------------------------------
--R writeObj : (SubSpace(3,DoubleFloat),String) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- Export3D.help ---

====================================================================
Export3D examples
====================================================================

This package provides support for exporting SubSpace and ThreeSpace structures to files.

See Also:
o )show Export3D
Export3D (EXP3D)

-- package EXP3D Export3D --

)abbrev package EXP3D Export3D
++ Author: Martin Baker
++ Date: June, 2010
++ Description:
++ This package provides support for exporting SubSpace and
++ ThreeSpace structures to files.

EF ==> Expression Float
SBF ==> SegmentBinding Float
DF ==> DoubleFloat
I  ==> Integer
PI  ==> PositiveInteger
NNI ==> NonNegativeInteger
STR ==> String

Export3D(): with

   writeObj:(SubSpace(3,DoubleFloat),String) -> Void
   ++ writes 3D SubSpace to a file in Wavefront (.OBJ) format

== add
import List List NNI

-- return list of indexes
-- assumes subnodes are leaves containing index
faceIndex(subSp: SubSpace(3,DoubleFloat)):List NNI ==
  faceIndexList:List NNI := []
  for poly in children(subSp) repeat
    faceIndexList := cons(extractIndex(poly),faceIndexList)
  reverse faceIndexList

-- called if this component contains a single polygon
-- write out face information for Wavefront (.OBJ) 3D file format
-- one face per line, represented by list of vertex indexes
writePolygon(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
  faceIndexList:List NNI := []
  for curve in curves repeat
    faceIndexList := append(faceIndexList,faceIndex(curve))
  -- write out face information for Wavefront (.OBJ) 3D file format
  -- one face per line, represented by list of vertex indexes
  s:String := "f 
  for i in faceIndexList repeat
    s:=concat(s,string(i))$String
    s:=concat(s," ")$String
  writeLine!(f1,s)

-- called if this component contains a mesh, the mesh will be rendered
-- as quad polygons.
-- write out face information for Wavefront (.OBJ) 3D file format
-- one face per line, represented by list of vertex indexes
writeMesh(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
  meshIndexArray:List List NNI := []
  for curve in curves repeat
    -- write out face information for Wavefront (.OBJ) 3D file format
    -- one face per line, represented by list of vertex indexes
    meshIndexArray := cons(faceIndex(curve),meshIndexArray)
  meshIndexArray := reverse meshIndexArray
  rowLength := #meshIndexArray
  colLength := #(meshIndexArray.1)
  for i in 1..(rowLength-1) repeat
    for j in 1..(colLength-1) repeat
      --s1:String := concat["row ",string(i)," col ",string(j)]
      --writeLine!(f1,s1)
      s:String := concat ["f ",string((meshIndexArray.i).j)," ",
        string((meshIndexArray.(i+1)).j)," ",
        string((meshIndexArray.(i+1)).(j+1))," ",
        string((meshIndexArray.i).(j+1))]
      writeLine!(f1,s)

toString(d : DoubleFloat) : String ==
  unparse(convert(d)@InputForm)

-- this writes SubSpace geometry to Wavefront (.OBJ) 3D file format
-- requires SubSpace to contain 3 or 4 dimensional points over DoubleFloat
-- to export a function plot try:
-- writeObj(subspace(makeObject(x*x-y*y,x=-1..1,y=-1..1)),"myfile.obj")
-- colour dimension is ignored
-- no normals or texture data is generated
writeObj(subSp: SubSpace(3,DoubleFloat), filename:String):Void ==
f1:TextFile:=open(filename::FileName,"output")
writeLine!(f1,"# mesh generated by axiom")
-- write vertex data
verts := pointData(subSp)
for v in verts repeat
  if v < 3 then error "Can't write OBJ file from 2D points"
writeLine!(f1,concat(["v ", toString(v.1), " ",
    toString(v.2), " ", toString(v.3)]]$String)
for component in children(subSp) repeat
  curves := children(component)
  if #curves < 2 then
    sayTeX$Lisp "Can't write point or curve to OBJ file"
    --writeLine!(f1,"new component")
  if #curves > 1 then
    if numberOfChildren(curves.1) = 1 then writePolygon(f1,curves)
    if numberOfChildren(curves.1) > 1 then writeMesh(f1,curves)
  close! f1

— EXP3D.dotabb —

"EXP3D" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXP3D"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"EXP3D" -> "STRING"

package E04AGNT e04AgentsPackage

— e04AgentsPackage.input —

)set break resume
)sys rm -f e04AgentsPackage.output
)spool e04AgentsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
e04AgentsPackage is a package of numerical agents to be used to investigate attributes of an input function so as to decide the measure of an appropriate numerical optimization routine.

See Also:
o )show e04AgentsPackage

Exports:
changeNameToObjf  expenseOfEvaluation  finiteBound  linear?  linearMatrix
linearPart  nonLinearPart  optAttributes  quadratic?  simpleBounds?
sortConstraints  splitLinear  sumOfSquares  varList  variables

--- package E04AGNT e04AgentsPackage ---

)abbrev package E04AGNT e04AgentsPackage
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: June 1996
++ Description:
++ \axiomType{e04AgentsPackage} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \axiomFun{measure} of an appropriate numerical optimization routine.

e04AgentsPackage(): E == I where
MDF ==> Matrix DoubleFloat
VEDF ==> Vector Expression DoubleFloat
EDF ==> Expression DoubleFloat
EFI ==> Expression Fraction Integer
PFI ==> Polynomial Fraction Integer
FI ==> Fraction Integer
F ==> Float
DF ==> DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
LOCDF ==> List OrderedCompletion DoubleFloat
LEDF ==> List Expression DoubleFloat
PDF ==> Polynomial DoubleFloat
LDF ==> List DoubleFloat
INT ==> Integer
NNI ==> NonNegativeInteger
LS ==> List Symbol
EF2 ==> ExpressionFunctions2
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
E ==> with
  finiteBound:(LOCDF,DF) -> LDF
  ++ finiteBound(l,b) replaces all instances of an infinite entry in
  ++ \axiom{l} by a finite entry \axiom{b} or \axiom{-b}.
  sortConstraints:NOA -> NOA
  ++ sortConstraints(args) uses a simple bubblesort on the list of
  ++ constraints using the degree of the expression on which to sort.
  ++ Of course, it must match the bounds to the constraints.
  sumOfSquares:EDF -> Union(EDF, "failed")
  ++ sumOfSquares(f) returns either an expression for which the square is
  ++ the original function of "failed".
  splitLinear:EDF -> EDF
  ++ splitLinear(f) splits the linear part from an expression which it
  ++ returns.
  simpleBounds?:LEDFF -> Boolean
  ++ simpleBounds?(l) returns true if the list of expressions l are
  ++ simple.
  linear?:LEDFF -> Boolean
  ++ linear?(l) returns true if all the bounds l are either linear or
  ++ simple.
  linear?:EDF -> Boolean
++ linear?(e) tests if \(\text{axiom}(e)\) is a linear function.
linearMatrix:(LEDF, NNI) -> MDF
++ linearMatrix(l,n) returns a matrix of coefficients of the linear
++ functions in \(\text{axiom}(l)\). If l is empty, the matrix has at least one
++ row.
linearPart:LEDF -> LEDF
++ linearPart(l) returns the list of linear functions of \(\text{axiom}(l)\).
nonLinearPart:LEDF -> LEDF
++ nonLinearPart(l) returns the list of non-linear functions of l.
quadratic?:EDF -> Boolean
++ quadratic?(e) tests if \(\text{axiom}(e)\) is a quadratic function.
variables:LSA -> LS
++ variables(args) returns the list of variables in \(\text{axiom}(\text{args.lfn})\)
varList:(EDF,NNI) -> LS
++ varList(e,n) returns a list of \(\text{axiom}(n)\) indexed variables with name
++ as in \(\text{axiom}(e)\).
changeNameToObjf:(Symbol,Result) -> Result
++ changeNameToObjf(s,r) changes the name of item \(\text{axiom}(s)\) in \(\text{axiom}(r)\)
++ to objf.
expenseOfEvaluation:LSA -> F
++ expenseOfEvaluation(o) returns the intensity value of the
++ cost of evaluating the input set of functions. This is in terms
++ of the number of ‘operational units’. It returns a value
++ in the range \([0,1]\).
optAttributes:Union(noa:NOA,lsa:LSA) -> List String
++ optAttributes(o) is a function for supplying a list of attributes
++ of an optimization problem.

I ==> add

import ExpertSystemToolsPackage, ExpertSystemContinuityPackage

sumOfSquares2:EFI -> Union(EFI,"failed")
nonLinear?:EDF -> Boolean
finiteBound2:(OCDF,DF) -> DF
functionType:EDF -> String

finiteBound2(a:OCDF,b:DF):DF ==
not finite?(a) =>
  positive?(a) => b
- b
retract(a)@DF

finiteBound(l:LOCDF,b:DF):LDF == [finiteBound2(i,b) for i in l]

sortConstraints(args:NOA):NOA ==
  Args := copy args
c:LEDF := Args.cf
l:LOCDF := Args.lb
u:LOCDF := Args.ub
\[
m: \text{INT} := (# c) - 1
\]
\[
n: \text{INT} := (# l) - m
\]
for \( j \) in \( m..1 \) by \(-1\) repeat
  for \( i \) in \( 1..j \) repeat
    \( s: \text{EDF} := c.i \)
    \( t: \text{EDF} := c.(i+1) \)
    if linear?(t) and (nonLinear?(s) or quadratic?(s)) then
      \( \text{swap}!(c,i,i+1)$\text{LEDF} \)
      \( \text{swap}!(l,n+i-1,n+i)$\text{LDCDF} \)
      \( \text{swap}!(u,n+i-1,n+i)$\text{LDCDF} \)

Args

changeNameToObjf(s: Symbol, r: Result): Result ==
a := \text{remove}!(s, r)$\text{Result}
a case Any =>
  \( \text{insert}!([\text{objf}@\text{Symbol}, a], r)$\text{Result} \)
r
r

sum(a: EDF, b: EDF): EDF == a + b

variables(args: LSA): LS == \text{variables}((\text{reduce}(\text{sum}, (\text{args}.lfn))))

sumOfSquares(f: EDF): \text{Union}(EDF, "failed") ==
e := edf2efi(f)
s: \text{Union}(EFI, "failed") := \text{sumOfSquares2}(e)
s case EFI =>
  \( \text{map}(\text{fi2df}, s)$\text{EF2(FI,DF)} \)
"failed"

sumOfSquares2(f: EFI): \text{Union}(EFI, "failed") ==
p := \text{retractIfCan}(f)@\text{Union}(PFI, "failed")
p case PFI =>
  \( \text{r} := \text{squareFreePart}(p)$\text{PFI} \)
  \( \text{(p=r)}@\text{Boolean} \Rightarrow "\text{failed}" \)
  \( \text{tp} := \text{totalDegree}(p)$\text{PFI} \)
  \( \text{tr} := \text{totalDegree}(r)$\text{PFI} \)
  \( t := \text{tp} \text{quo} \text{tr} \)
  \( \text{found} := \text{false} \)
  \( q := r \)
  for \( i \) in \( 2..t \) by \(2\) repeat
    \( s := q^2 \)
    \( (s=p)@\text{Boolean} \Rightarrow \) found := true
    \( \text{leave} \)
  \( q := r^i \)
  if found then
    \( q :: EFI \)
  else
    "failed"
"failed"

splitLinear(f: EDF): EDF ==
  out := 0$EDF
  (l := isPlus(f)$EDF) case LEDF =>
    for i in l repeat
      if not quadratic? i then
        out := out + i
    out
  out

edf2pdf(f: EDF): PDF == (retract(f)@PDF)$EDF

varList(e: EDF, n: NNI): LS ==
  s := name(first(variables(edf2pdf(e))$PDF)$LS)$Symbol
  [subscript(s, [t::OutputForm]) for t in expand([1..n])$Segment(Integer)]

functionType(f: EDF): String ==
  n := #(variables(f))$EDF
  p := (retractIfCan(f)@Union(PDF, "failed"))$EDF
  p case PDF =>
    d := totalDegree(p)$PDF
    -- one?(n*d) => "simple"
    (n*d) = 1 => "simple"
    -- one?(d) => "linear"
    (d = 1) => "linear"
    (d=2)$Boolean => "quadratic"
    "non-linear"
"non-linear"

simpleBounds?(l: LEDF): Boolean ==
  a := true
  for e in l repeat
    not (functionType(e) = "simple")$Boolean =>
      a := false
      leave
    a

simple?(e: EDF): Boolean == (functionType(e) = "simple")$Boolean

linear?(e: EDF): Boolean == (functionType(e) = "linear")$Boolean

quadratic?(e: EDF): Boolean == (functionType(e) = "quadratic")$Boolean

nonLinear?(e: EDF): Boolean == (functionType(e) = "non-linear")$Boolean

linear?(l: LEDF): Boolean ==
  a := true
  for e in l repeat
    s := functionType(e)
(s = "quadratic")@Boolean or (s = "non-linear")@Boolean =>
a := false
leave
a

simplePart(l:LEDF):LEDF == [i for i in l | simple?(i)]
linearPart(l:LEDF):LEDF == [i for i in l | linear?(i)]
nonLinearPart(l:LEDF):LEDF ==
  [i for i in l | not linear?(i) and not simple?(i)]

linearMatrix(l:LEDF, n:NNI):MDF ==
  empty?(l) => mat([],n)
  L := linearPart l
  M := zero(max(1,# L)$NNI,n)$MDF
  vars := varList(first(l)$LEDF,n)
  row:INT := 1
  for a in L repeat
    for j in monomials(edf2pdf(a))$PDF repeat
      col:INT := 1
      for c in vars repeat
        if ((first(variables(j)$PDF)$LS)=c)@Boolean then
          M(row,col):= first(coefficients(j)$PDF)$LDF
          col := col+1
      row := row + 1
  M

expenseOfEvaluation(o:LSA):F ==
  expenseOfEvaluation(vector(copy o.lfn)$VEDF)

optAttributes(o:Union(noa:NOA,lsa:LSA)):List String ==
o case noa =>
  n := o.noa
  s1:String := "The object function is " functionType(n.fn)
  if empty?(n.lb) then
    s2:String := "There are no bounds on the variables"
  else
    s2:String := "There are simple bounds on the variables"
  c := n.cf
  if empty?(c) then
    s3:String := "There are no constraint functions"
  else
    t := #(c)
    lin := #(linearPart(c))
    nonlin := #(nonLinearPart(c))
    s3:String := "There are " string(lin)$String " linear and "
      string(nonlin)$String " non-linear constraints"
  [s1,s2,s3]
  l := o.lsa
s := "non-linear"
if linear?(l.lfn) then
  s := "linear"
["The object functions are " s]
Chapter 7

Chapter F

package FACTFUNC FactoredFunctions

— FactoredFunctions.input —

)set break resume
)sys rm -f FactoredFunctions.output
)spool FactoredFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactoredFunctions
--E 1

)spool
)lisp (bye)

— FactoredFunctions.help —

====================================================================
FactoredFunctions examples
====================================================================

Computes various functions on factored arguments.

See Also:
o )show FactoredFunctions

637
FactoredFunctions (FACTFUNC)

Exports:
log    nthRoot

--- package FACTFUNC FactoredFunctions ---

)abbrev package FACTFUNC FactoredFunctions
++ Author: Manuel Bronstein
++ Date Created: 2 Feb 1988
++ Date Last Updated: 25 Jun 1990
++ Description:
++ computes various functions on factored arguments.
-- not visible to the user

FactoredFunctions(M:IntegralDomain): Exports == Implementation where
N ==> NonNegativeInteger

Exports == with
nthRoot: (Factored M,N) -> Record(exponent:N,coef:M,radicand:List M)
  ++ nthRoot(f, n) returns \spad{\{(p, r, [r1,\ldots,rm]\}\} such that
  ++ the nth-root of f is equal to \spad{r \ast \sqrt[p]{r1 \ast \ldots \ast rm}},
  ++ where r1,\ldots,rm are distinct factors of f,
  ++ each of which has an exponent smaller than p in f.
log : Factored M -> List Record(coef:N, logand:M)
  ++ log(f) returns \spad{\{(a1,b1),\ldots,(am,bm)\} such that
  ++ the logarithm of f is equal to \spad{a1\log(b1) + \ldots + am\log(bm)}.

Implementation ==> add
nthRoot(ff, n) ==

coeff:M := 1
-- radi:List(M) := (one? unit ff => empty(); [unit ff])
radi:List(M) := (((unit ff) = 1) => empty(); [unit ff])
lf := factors ff

d:N :=
  empty? radi => gcd(concat(n, [t.exponent::N for t in lf]))::N
  1
n := n quo d
for term in lf repeat
  qr := divide(term.exponent::N quo d, n)
  coeff := coeff * term.factor ** qr.quotient
  not zero?(qr.remainder) =>
    radi := concat_!(radi, term.factor ** qr.remainder)
[n, coeff, radi]

log ff ==
  ans := unit ff
  concat([1, unit ff],
    [[term.exponent::N, term.factor] for term in factors ff])

package FR2 FactoredFunctions2

--- FACTFUNC.dotabb ---

"FACTFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTFUNC"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FACTFUNC" -> "ALGEBRA"

---

package FR2 FactoredFunctions2

--- FactoredFunctions2.input ---

)set break resume
)sys rm -f FactoredFunctions2.output
)spool FactoredFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 6
double(x) == x + x
--R
--R Type: Void
f := factor(720)

map(double,f)

makePoly(b) == x + b

g := map(makePoly,f)

nthFlag(g,1)
FactoredFunctions2 contains functions that involve factored objects whose underlying domains may not be the same. For example, map might be used to coerce an object of type Factored(Integer) to Factored(Complex(Integer)).

The FactoredFunctions2 package implements one operation, map, for applying an operation to every base in a factored object and to the unit.

\[
double(x) = x + x
\]

Type: Void

\[
f := \text{factor}(720)
\]

\[
\begin{align*}
4 & 2 \\
2 & 3 & 5
\end{align*}
\]

Type: Factored Integer

Actually, the map operation used in this example comes from Factored itself, since double takes an integer argument and returns an integer result.

\[
\text{map}(\text{double}, f)
\]

\[
\begin{align*}
4 & 2 \\
2 & 4 & 6 & 10
\end{align*}
\]

Type: Factored Integer

If we want to use an operation that returns an object that has a type different from the operation's argument, the map in Factored cannot be used and we use the one in FactoredFunctions2.

\[
\text{makePoly}(b) = x + b
\]

\[
\begin{align*}
4 & 2 \\
(x + 1)(x + 2)(x + 3)(x + 5)
\end{align*}
\]

Type: Factored Polynomial Integer

In fact, the "2" in the name of the package means that we might be using factored objects of two different types.

\[
g := \text{map}(\text{makePoly}, f)
\]

It is important to note that both versions of map destroy any information known about the bases (the fact that they are prime, for instance).

The flags for each base are set to "nil" in the object returned by map.
\texttt{nthFlag(g,1)}
\texttt{"nil"}

Type: \texttt{Union("nil",...)}

See Also:
- \texttt{)help Factored}
- \texttt{)show FactoredFunctions2}

---

\textbf{FactoredFunctions2 (FR2)}

\begin{center}
\begin{tikzpicture}
  \node[anchor=north west,inner sep=0] (image) at (0,0) {
    \includegraphics[width=\textwidth]{fr2.png}
  };
  \begin{scope}[x={(image.south east)},y={(image.north west)}]
    \node[above right] at (0.5,0.5) {FR2};
    \node[below right] at (0.5,0) {ALGEBRA};
  \end{scope}
\end{tikzpicture}
\end{center}

Exports:
\texttt{map}

---

\texttt{)abbrev package FR2 FactoredFunctions2}
++ Author: Robert S. Sutor
++ Date Created: 1987
++ Description:
++ \texttt{FactoredFunctions2} contains functions that involve
++ factored objects whose underlying domains may not be the same.
++ For example, \texttt{map} might be used to coerce an object of
++ type \texttt{Factored(Integer)} to
++ \texttt{Factored(Complex(Integer))}.

\texttt{FactoredFunctions2}(R, S): Exports == Implementation where
R: IntegralDomain
S: IntegralDomain

Exports => with
map: (R -> S, Factored R) -> Factored S
++ map(fn,u) is used to apply the function \userfun{fn} to every
++ factor of \spadvar{u}. The new factored object will have all its
++ information flags set to "nil". This function is used, for
++ example, to coerce every factor base to another type.

Implementation ==> add
map(func, f) ==
  func(unit f) *
  _.*/[nilFactor(func(g.factor), g.exponent) for g in factors f]

---

package FRUTIL FactoredFunctionUtilities

--- FactoredFunctionUtilities.input ---

)set break resume
)sys rm -f FactoredFunctionUtilities.output
)spool FactoredFunctionUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactoredFunctionUtilities
--E 1

)spool
)lisp (bye)

---

--- FactoredFunctionUtilities.help ---

====================================================================
FactoredFunctionUtilities examples
====================================================================
FactoredFunctionUtilities implements some utility functions for
manipulating factored objects.

See Also:
o)show FactoredFunctionUtilities

FactoredFunctionUtilities (FRUTIL)

Exports:
mergeFactors refine

--- package FRUTIL FactoredFunctionUtilities ---

)abbrev package FRUTIL FactoredFunctionUtilities
++ Description:
++ spadtype{FactoredFunctionUtilities} implements some utility
++ functions for manipulating factored objects.

FactoredFunctionUtilities(R):Exports == Implementation where
  R: IntegralDomain
  FR ==> Factored R

Exports ==> with
  refine: (FR, R-> FR) -> FR
    ++ refine(u,fn) is used to apply the function \userfun{fn} to
    ++ each factor of \spadvar{u} and then build a new factored
    ++ object from the results. For example, if \spadvar{u} were
    ++ created by calling \spad{nilFactor(10,2)} then
\spad{refine(u,factor)} would create a factored object equal
++ to that created by \spad{factor(100)} or
++ \spad{primeFactor(2,2) * primeFactor(5,2)}.

mergeFactors: (FR,FR) -> FR
++ mergeFactors(u,v) is used when the factorizations of \spadvar{u}
++ and \spadvar{v} are known to be disjoint, e.g. resulting from a
++ content/primitive part split. Essentially, it creates a new
++ factored object by multiplying the units together and appending
++ the lists of factors.

Implementation ==> add
fg: FR
func: R -> FR
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF ==> Record(flg: fUnion, fctr: R, xpnt: Integer)
mergeFactors(f,g) ==
makeFR(unit(f)*unit(g),append(factorList f,factorList g))
refine(f, func) ==
u := unit(f)
l: List FF := empty()
for item in factorList f repeat
  fitem := func item.fctr
  u := u*unit(fitem) ** (item.xpnt :: NonNegativeInteger)
  if item.xpnt = 1 then
    l := concat(factorList fitem,l)
  else l := concat([v.flg,v.fctr,v.xpnt*item.xpnt]
                   for v in factorList fitem],l)
makeFR(u,l)

———

— FRUTIL.dotabb ——

"FRUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRUTIL"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FRUTIL" -> "ALGEBRA"

———

package FACUTIL FactoringUtilities

— FactoringUtilities.input —
FactoringUtilities (FACUTIL)

This package provides utilities used by the factorizers which operate on polynomials represented as univariate polynomials with multivariate coefficients.

See Also:
  o )show FactoringUtilities

---

FactoringUtilities.help —

====================================================================
FactoringUtilities examples
====================================================================

This package provides utilities used by the factorizers which operate on polynomials represented as univariate polynomials with multivariate coefficients.

See Also:
  o )show FactoringUtilities

---

FactoringUtilities (FACUTIL)
Exports:

- completeEval
- degree
- lowerPolynomial
- normalDeriv
- raisePolynomial
- ran
- variables

--- package FACUTIL FactoringUtilities ---

\texttt{(}\texttt{add}\texttt{)}
lowerPolynomial(f: SUP P) : SUP R ==
  zero? f => 0$SUP(R)
  monomial(ground leadingCoefficient f, degree f)$SUP(R) +
  lowerPolynomial(reductum f)

raisePolynomial(u: SUP R) : SUP P ==
  zero? u => 0$SUP(P)
  monomial(leadingCoefficient(u)::P, degree u)$SUP(P) +
  raisePolynomial(reductum u)

completeEval(f:SUP P,lvar:List OV,lval:List R) : SUP R ==
  zero? f => 0$SUP(R)
  monomial(ground eval(leadingCoefficient f,lvar,lval),degree f)$SUP(R) +
  completeEval(reductum f,lvar,lval)

degree(f:SUP P,lvar:List OV) : List NNI ==
  coefs := coefficients f
  ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]

variables(f:SUP P) : List OV ==
  "setUnion"/[variables cf for cf in coefficients f]

if R has FiniteFieldCategory then
  ran(k:Z):R == random()$R
else
  ran(k:Z):R == (random(2*k+1)$Z -k)::R

-- Compute the normalized m derivative
normalDeriv(f:SUP P,m:Z) : SUP P==
  (n1:Z:=degree f) < m => 0$SUP(P)
  n1=m => (leadingCoefficient f)::SUP(P)
  k:=binomial(n1,m)
  ris:=SUP:=0$SUP(P)
  n:Z:=n1
  while n>= m repeat
    while n1>n repeat
      k:=(k*(n1-m)) quo n1
      n1:=n1-1
      ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
      f:=reductum f
      n:=degree f
    ris
  ---

| FACUTIL.dotabb |

"FACUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACUTIL"]
package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

--- FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.input ---

)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)spool FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

--- FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

---_factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
---_factorSqFree : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))

---E 1

)spool
)lisp (bye)

---

--- FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.help ---

====================================================================
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber examples
====================================================================
Part of the Package for Algebraic Function Fields in one variable PAFF

See Also:
- )show FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

---

FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber (FACTEXT)

---

package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

)abbrev package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber(K):Exports == Implementation where
K:PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory
SUP ==> SparseUnivariatePolynomial
Q ==> PseudoAlgebraicClosureOfRationalNumber
NNI ==> NonNegativeInteger
UP ==> SUP(K)
UPUP ==> SUP(SUP(K))
FACTRNQ ==> FactorisationOverPseudoAlgebraicClosureOfRationalNumber(Q)
Exports ==> with
factor: (UP,K) -> Factored UP
factorSqFree: (UP,K) -> Factored UP

Implementation ==> add
up2Rat: UP -> SUP(Q)
rat2up: SUP(Q) -> UP

factRat: UP -> Factored UP
liftPoly: (UP, K) -> UPUP

liftDefPoly: UP -> UPUP

norm: (UP, K) -> UP

factParPert: (UP,K,K) -> Factored UP

trans: (UP, K) -> UP

swapCoefWithVar: (UP, NNI) -> UPUP

frRat2frUP: Factored SUP(Q) -> Factored UP

factor(pol,a)==
polSF:= squareFree pol
reduce("*", [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI) |
for fr in factorList polSF ], 1)

factorSqFree(pol,a)==
ratPol:SUP(Q)
aa:Q
ground? a =>
aa:= retract(a)@Q
ratPol:= up2Rat pol
frRat2frUP factor(ratPol,aa)$FACTRNQ::Factored UP
nPol:= norm(pol,a)
ta:=previousTower a
factN := factor( nPol , ta )
lfactnP0l:= factorList factN
G:UP:=1
L: Factored UP:= 1
for fr in lfactnP0l repeat
G:= gcd( [ pol , fr.fctr ] )
pol:= pol quo$UP G
if one? fr.xpnt then
L := L * flagFactor( G, 1 ,"prime")$Factored(UP)
else
L := L * factParPert( G, a, a )
L

factParPert(pol, a, b)==
polt:=trans(pol,b)
frpol:= factorList factor(polt,a)
sl:= [ fr.fctr for fr in frpol ]
slt:= [ trans(p, -b) for p in sl ]
nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) 
       for p in slt for fr in frpol ]
reduce("*", nfrpol)

frRat2frUP(fr)==
frpol:= factorList fr
sl:= [ fr.fctr for fr in frpol ]
slt:= [ rat2up p for p in sl ]
nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) 
       for p in slt for fr in frpol ]
reduce("*", nfrpol)

up2Rat(pol)==
zero?(pol) => 0
d:=degree pol
a:Q:= retract(leadingCoefficient pol)@Q
monomial(a,d)$SUP(Q) + up2Rat(reductum pol)

rat2up(pol)==
zero?(pol) => 0
d:=degree pol
a:K:=(leadingCoefficient pol) :: K
monomial(a,d)$UP + rat2up(reductum pol)

trans(pol,a)==
zero? pol => 0
lc:=leadingCoefficient pol
d:=degree pol
lc*(monomial(1,1)$UP + monomial(-a,0)$UP)**d + trans(reductum pol,a)

liftDefPoly(pol)==
zero?(pol) => 0
lc:= leadingCoefficient pol
d:= degree pol
monomial( monomial(lc,0)$UP , d )$UPUP + liftDefPoly reductum pol

norm(pol,a)==
lpol:=liftPoly(pol,a)
defPol:=definingPolynomial a
ldefPol:=liftDefPoly defPol
resultant(ldefPol,lpol)

swapCoefWithVar(coef,n)==
ground? coef =>
monomial( monomial( retract coef , n)$SUP(K) , 0)$UPUP
lcoef:=leadingCoefficient(coef)
d:=degree(coef)
package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber

---

FactorisationOverPseudoAlgebraicClosureOfRationalNumber.input

)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfRationalNumber.output
)spool FactorisationOverPseudoAlgebraicClosureOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfRationalNumber
--R
--R FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K: PseudoAlgebraicClosureOfRationalNumberCategory) is a package constructor
--R Abbreviation for FactorisationOverPseudoAlgebraicClosureOfRationalNumber is FACTRN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTRN
--R
--R------------------------------------- Operations -------------------------------------
--R factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
FactorisationOverPseudoAlgebraicClosureOfRationalNumber (FACTRN)

Exports:
factor factorSqFree

— package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber —
)abbrev package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K):Exports ==
    Implementation where
    K: PseudoAlgebraicClosureOfRationalNumberCategory
    SUP ==> SparseUnivariatePolynomial
    Q ==> Fraction Integer
    NNI ==> NonNegativeInteger
    UP ==> SUP(K)
    UPUP ==> SUP(SUP(K))

Exports ==> with
    factor: (UP,K) -> Factored UP
    factorSqFree: (UP,K) -> Factored UP

Implementation ==> add
    up2Rat: UP -> SUP(Q)
    rat2up: SUP(Q) -> UP
    factRat: UP -> Factored UP
    liftPoly: (UP, K) -> UPUP
    liftDefPoly: UP -> UPUP
    norm: (UP, K) -> UP
    factParPert: ( UP,K,K) -> Factored UP
    trans: (UP, K) -> UP
    swapCoefWithVar: ( UP , NNI) -> UPUP
    frRat2frUP: Factored SUP(Q) -> Factored UP

factor(pol,a)==
    polSF:= squareFree pol
    reduce("*", _,
        [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI) _
            for fr in factorList polSF], 1)

factorSqFree(pol,a)==
    ratPol: SUP(Q)
    ground? a =>
        ratPol:= up2Rat pol
        frRat2frUP factor( ratPol )$RationalFactorize( SUP(Q) ) :: Factored UP
    nPol:= norm(pol,a)
ta:=previousTower a
factN := factor( nPol , ta )
lfactnPol:= factorList factN
G:UP:=1
L: Factored UP:= 1
for fr in lfactnPol repeat
   G:= gcd( [ pol , fr.fctr ] )
pol:= pol quo$UP G
if one? fr.xpnt then
   L := L * flagFactor( G, 1 ,"prime")$Factored(UP)
else
   L := L * factParPert( G, a, a )
L

factParPert(pol, a, b)==
polt:=trans(pol,b)
frpol:= factorList factor(polt,a)
sl:= [ fr.fctr for fr in frpol ]
slt:= [ trans(p , -b) for p in sl ]
nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
   for p in slt for fr in frpol ]
reduce("*", nfrpol)

frRat2frUP(fr)==
frpol:= factorList fr
sl:= [ fr.fctr for fr in frpol ]
slt:= [ rat2up p for p in sl ]
nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
   for p in slt for fr in frpol ]
reduce("*", nfrpol)

up2Rat(pol)==
zero?(pol) => 0
d:=degree pol
a:Q:= retract(leadingCoefficient pol)@Q
monomial(a,d)$SUP(Q) + up2Rat(reductum pol)

rat2up(pol)==
zero?(pol) => 0
d:=degree pol
a:K:=(leadingCoefficient pol) :: K
monomial(a,d)$UP + rat2up(reductum pol)

trans(pol,a)==
zero? pol => 0
lc:=leadingCoefficient pol
d:=degree pol
lc*( monomial(1,1)$UP + monomial(-a,0)$UP )**d + trans(reductum pol , a)
liftDefPoly(pol) ==
  zero?(pol) => 0
  lc := leadingCoefficient pol
  d := degree pol
  monomial( monomial(lc, 0)$UP, d )$UPUP + liftDefPoly reductum pol

norm(pol, a) ==
  lpol := liftPoly(pol, a)
  defPol := definingPolynomial a
  ldefPol := liftDefPoly defPol
  resultant(ldefPol, lpol)

swapCoefWithVar(coef, n) ==
  ground? coef =>
  monomial( monomial( retract coef, n)$SUP(K), 0)$UPUP
  lcoef := leadingCoefficient(coef)
  d := degree(coef)
  monomial( monomial(lcoef, n)$SUP(K), d)$UPUP + 
    swapCoefWithVar( reductum coef, n )

liftPoly(pol, a) ==
  zero? pol => 0
  lcoef := leadingCoefficient pol
  n := degree pol
  liftCoef := lift(lcoef, a)$K
  swapCoefWithVar(liftCoef, n) + liftPoly( reductum pol, a )
FGLMIfCanPackage (FGLMICPK)

Exports:
   fglmIfCan groebner zeroDimensional?

This is just an interface between several packages and domains. The goal is to compute lexicographical Groebner bases of sets of polynomial with type Polynomial R by the FGLM algorithm if this is possible (i.e. if the input system generates a zero-dimensional ideal).

See Also:
o )show FGLMIfCanPackage
package FGLMICPK FGLMifCanPackage

)abbrev package FGLMICPK FGLMifCanPackage
++ Author: Marc Moreno Maza
++ Date Created: 08/02/1999
++ Date Last Updated: 08/02/1999
++ Description:
++ This is just an interface between several packages and domains.
++ The goal is to compute lexicographical Groebner bases
++ of sets of polynomial with type \spadtype{Polynomial R}
++ by the FGLM algorithm if this is possible (i.e.
++ if the input system generates a zero-dimensional ideal).

FGLMifCanPackage(R,ls): Exports == Implementation where
  R: GcdDomain
  ls: List Symbol
  V ==> OrderedVariableList ls
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  Q1 ==> Polynomial R
  Q2 ==> HomogeneousDistributedMultivariatePolynomial(ls,R)
  Q3 ==> DistributedMultivariatePolynomial(ls,R)
  E2 ==> HomogeneousDirectProduct(#ls,NonNegativeInteger)
  E3 ==> DirectProduct(#ls,NonNegativeInteger)
  poltopol ==> PolToPol(ls, R)
  lingrobpack ==> LinGroebnerPackage(ls,R)
  groebnerpack2 ==> GroebnerPackage(R,E2,V,Q2)
  groebnerpack3 ==> GroebnerPackage(R,E3,V,Q3)

Exports == with

  zeroDimensional?: List(Q1) -> B
  ++ \axiom{zeroDimensional?(lq1)} returns true iff
  ++ \axiom{lq1} generates a zero-dimensional ideal
  ++ w.r.t. the variables of \axiom{ls}.
  fglmIfCan: List(Q1) -> Union(List(Q1), "failed")
  ++ \axiom{fglmIfCan(lq1)} returns the lexicographical Groebner
  ++ basis of \axiom{lq1} by using the FGLM strategy,
  ++ if \axiom{zeroDimensional?(lq1)} holds.
  groebner: List(Q1) -> List(Q1)
  ++ \axiom{groebner(lq1)} returns the lexicographical Groebner
  ++ basis of \axiom{lq1}. If \axiom{lq1} generates a zero-dimensional
  ++ ideal then the FGLM strategy is used, otherwise
  ++ the Sugar strategy is used.

Implementation == add

  zeroDim?(lq2: List Q2): Boolean ==
  lq2 := groebner(lq2)$groebnerpack2
empty? lq2 => false
#lq2 < #ls => false
lv: List(V) := [(variable(s)$V)::V for s in ls]
for q2 in lq2 while not empty?(lv) repeat
  m := leadingMonomial(q2)
  x := mainVariable(m)::V
  if ground?(leadingCoefficient(univariate(m,x))) then
    lv := remove(x, lv)
empty? lv

zeroDimensional?(lq1: List(Q1)): Boolean ==
lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
zeroDim?(lq2)

fglmIfCan(lq1:List(Q1)): Union(List(Q1),"failed") ==
lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
lq2 := groebner(lq2)$groebnerpack2
not zeroDim?(lq2) => "failed":Union(List(Q1),"failed")
lq3: List(Q3) := totolex(lq2)$lingrobpack
lq1 := [dmpToP(q3)$poltopol for q3 in lq3]
lq1::Union(List(Q1),"failed")
groebner(lq1:List(Q1)): List(Q1) ==
lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
lq2 := groebner(lq2)$groebnerpack2
not zeroDim?(lq2) =>
lq3: List(Q3) := [pToDmp(q1)$poltopol for q1 in lq1]
lq3 := groebner(lq3)$groebnerpack3
[dmpToP(q3)$poltopol for q3 in lq3]
lq3: List(Q3) := totolex(lq2)$lingrobpack
[dmpToP(q3)$poltopol for q3 in lq3]

-----------

"FGLMICPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FGLMICPK"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FGLMICPK" -> "DIRPCAT"
"FGLMICPK" -> "PFECAT"
package FORDER FindOrderFinite

--- FindOrderFinite.input ---

)set break resume
)sys rm -f FindOrderFinite.output
)spool FindOrderFinite.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show FindOrderFinite
-- E 1

)spool
)lisp (bye)

---

--- FindOrderFinite.help ---

====================================================================
FindOrderFinite examples
====================================================================

Finds the order of a divisor over a finite field

See Also:
o )show FindOrderFinite

---
FindOrderFinite (FORDER)

--- package FORDER FindOrderFinite ---

)abbrev package FORDER FindOrderFinite
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Finds the order of a divisor over a finite field

FindOrderFinite(F, UP, UPUP, R):Exports == Implementation where
F : Join(Finite, Field)
UP : UnivariatePolynomialCategory F
UPUP: UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F, UP, UPUP)

Exports ==> with
order: FiniteDivisor(F, UP, UPUP, R) -> NonNegativeInteger
++ order(x) undocumented
Implementation ==> add
order d ==
   dd := d := reduce d
   for i in 1.. repeat
      principal? dd => return(i::NonNegativeInteger)
   dd := reduce(d + dd)

FORDER.dotabb
"FORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORDER"]
package FAMR2 FiniteAbelianMonoidRingFunctions2

--- FiniteAbelianMonoidRingFunctions2.input ---

)set break resume
)sys rm -f FiniteAbelianMonoidRingFunctions2.output
)spool FiniteAbelianMonoidRingFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteAbelianMonoidRingFunctions2
--E 1

)spool
)lisp (bye)

---

--- FiniteAbelianMonoidRingFunctions2.help ---

====================================================================
FiniteAbelianMonoidRingFunctions2 examples
====================================================================

FiniteAbelianMonoidRing
The packages defined in this file provide fast fraction free rational interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

See Also:
o )show FiniteAbelianMonoidRingFunctions2
FiniteAbelianMonoidRingFunctions2 (FAMR2)

Exports:
map

--- package FAMR2 FiniteAbelianMonoidRingFunctions2 ---

)abbrev package FAMR2 FiniteAbelianMonoidRingFunctions2
++ Author: Martin Rubey
++ Description:
++ This package provides a mapping function for
++ \spad{FiniteAbelianMonoidRing}
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FiniteAbelianMonoidRingFunctions2(E: OrderedAbelianMonoid,
   R1: Ring,
   A1: FiniteAbelianMonoidRing(R1, E),
   R2: Ring,
   A2: FiniteAbelianMonoidRing(R2, E)) _

Exports == with

map: (R1 -> R2, A1) -> A2
++ \spad{map}(f, a) applies the map f to each coefficient in a. It is
++ assumed that f maps 0 to 0

Implementation == add

map(f: R1 -> R2, a: A1): A2 ==
if zero? a then 0$A2
else
  monomial(f leadingCoefficient a, degree a)$A2 + map(f, reductum a)

---
package FDIV2 FiniteDivisorFunctions2

--- FiniteDivisorFunctions2.input ---

)set break resume
)sys rm -f FiniteDivisorFunctions2.output
)spool FiniteDivisorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteDivisorFunctions2
--E 1

)spool
)lisp (bye)

--- FiniteDivisorFunctions2.help ---

====================================================================
FiniteDivisorFunctions2 examples
====================================================================

Lift a map to finite divisors.

See Also:
o )show FiniteDivisorFunctions2
FiniteDivisorFunctions2 (FDIV2)

Exports:

map

--- package FDIV2 FiniteDivisorFunctions2 ---

)abbrev package FDIV2 FiniteDivisorFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 19 May 1993
++ Description:
++ Lift a map to finite divisors.

FiniteDivisorFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):
Exports == Implementation where
  R1 : Field
  UP1 : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  F1 : FunctionFieldCategory(R1, UP1, UPUP1)
  R2 : Field
  UP2 : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2
  F2 : FunctionFieldCategory(R2, UP2, UPUP2)

Exports => with
  map: (R1 -> R2, FiniteDivisor(R1, UP1, UPUP1, F1)) ->
       FiniteDivisor(R2, UP2, UPUP2, F2)
    ++ map(f,d) \ undocumented{}

Implementation => add
  import UnivariatePolynomialCategoryFunctions2(R1,UP1,R2,UP2)
  import FunctionFieldCategoryFunctions2(R1,UP1,UPUP1,F1,R2,UP2,UPUP2,F2)
  import FractionalIdealFunctions2(UP1, Fraction UP1, UPUP1, F1,
                                 UP2, Fraction UP2, UPUP2, F2)

  map(f, d) =
rec := decompose d
  divisor map(f, rec.principalPart) +
  divisor map((s:UP1):UP2 +-> map(f,s), rec.id)

package FFFACTOR FiniteFieldFactorization

--- FiniteFieldFactorization.input ---

)set break resume
)sys rm -f FiniteFieldFactorization.output
)spool FiniteFieldFactorization.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldFactorization
--R FiniteFieldFactorization(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory(K)) is a package constructor
--R Abbreviation for FiniteFieldFactorization is FFFACTOR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFACTOR
--R
--R------------------------------- Operations ----------------------------------
--R factorCantorZassenhaus : (PolK,NonNegativeInteger) -> List(PolK)
--R factorSquareFree : PolK -> List(PolK)
--R factorUsingMusser : PolK -> Factored(PolK)
--R factorUsingYun : PolK -> Factored(PolK)
--R
--E 1

)spool
)lisp (bye)
CHAPTER 7. CHAPTER F

 FiniteFieldFactorization (FFFACTOR)

 )abbreviation package FFFACTOR FiniteFieldFactorization
 ++ Author: Gaetan Hache
 ++ Date Created: 17 nov 1992
 ++ Date Last Updated: March 2013 by Tim Daly
 ++ Description:
 ++ Part of the PAFF package
 FiniteFieldFactorization(K : FiniteFieldCategory,
 PolK : UnivariatePolynomialCategory(K)) : with

 factorSquareFree : PolK -> List(PolK)
factorCantorZassenhaus : (PolK, NonNegativeInteger) -> List(PolK)

factor : PolK -> Factored(PolK)

factorUsingYun : PolK -> Factored(PolK)

factorUsingMusser : PolK -> Factored(PolK)

irreducible? : PolK -> Boolean

== add

import FiniteFieldSquareFreeDecomposition(K, PolK)

p : NonNegativeInteger := characteristic()$K

p' : NonNegativeInteger := p quo 2 -- used for odd p : (p-1)/2

q : NonNegativeInteger := size()$K

q' : NonNegativeInteger := q quo 2 -- used for odd q : (q-1)/2

X : PolK := monomial(1, 1)

primeKdim : NonNegativeInteger := q_quo_p : NonNegativeInteger := q quo p ; e : NonNegativeInteger := 1
while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)
e

exp(P : PolK, n : NonNegativeInteger, R : PolK) : PolK ==
PP : PolK := P rem R ; Q : PolK := 1
repeat
  if odd?(n) then Q := Q * PP rem R
  (n := n quo 2) = 0 => leave
  PP := PP * PP rem R
return Q

pPowers(P : PolK) : PrimitiveArray(PolK) == -- P is monic
n := degree(P)
result : PrimitiveArray(PolK) := new(n, 1)
result(1) := Q1 := Q := exp(X, p, P)
for i in 2 .. n-1 repeat (Q1 := Q1*Q rem P ; result(i) := Q1)
return result

pExp(Q : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
Q' : PolK := 0
while Q' ~= 0 repeat
  Q' := Q' +primeFrobenius(leadingCoefficient(Q))*Xpowers(degree(Q))
  Q := reductum(Q)
return Q'

\( p\text{Trace}(Q : \text{PolK}, \, d : \text{NonNegativeInteger}, \, P : \text{PolK}, \)
\[ \text{Xpowers} : \text{PrimitiveArray} (\text{PolK}) ) : \text{PolK} == \]
\( Q : \text{PolK} := Q \text{ rem } P \)
\( \text{result} : \text{PolK} := Q \)
\( \text{for } i \text{ in } 1 \ldots d-1 \text{ repeat result := } Q + p\text{Exp(result, Xpowers)} \)
\( \text{return result rem } P \)

\( \text{random}(n : \text{NonNegativeInteger}) : \text{PolK} == \)
\( \text{repeat} \)
\( \text{if } (\text{deg} := (\text{random}(n)$\text{Integer})::\text{NonNegativeInteger}) > 0 \text{ then leave} \)
\( \text{repeat} \)
\( \text{if } (x : K := \text{random()}$K) \not= 0 \text{ then leave} \)
\( \text{result} : \text{PolK} := \)
\( \text{monomial}(x, \text{deg}) + +/[\text{monomial}(\text{random()}$K, i) \text{ for } i \text{ in } 0 \ldots \text{deg}-1] \)
\( \text{return result} \)

\( \text{internalFactorCZ}(P : \text{PolK}, \quad -- P \text{ monic-squarefree} \)
\[ \text{d} : \text{NonNegativeInteger}, \text{Xpowers} : \text{PrimitiveArray} (\text{PolK}) ) : \text{List} (\text{PolK}) == \]
\( \text{listOfFactors} : \text{List} (\text{PolK}) := [P] \)
\( \text{degree}(P) = d \Rightarrow \text{return listOfFactors} \)
\( \text{result} : \text{List} (\text{PolK}) := [] \)
\( \text{pDim} : \text{NonNegativeInteger} := d * \text{primeKdim} \)
\( Q : \text{PolK} := P \)
\( \text{repeat} \)
\( G := p\text{Trace}(\text{random} (\text{degree}(Q)), \text{pDim}, Q, \text{Xpowers}) \)
\( \text{if } p > 2 \text{ then } G := \text{exp}(G, p', Q) - 1 \)
\( Q1 := \text{gcd}(G, Q) ; \, d1 := \text{degree}(Q1) \)
\( \text{if } d1 > 0 \text{ and } d1 < \text{degree}(Q) \text{ then} \)
\( \text{listOfFactors} := \text{rest}(\text{listOfFactors}) \)
\( \text{if } d1 = d \text{ then result := cons}(Q1, \text{result}) \)
\( \quad \text{else listOfFactors := cons}(Q1, \text{listOfFactors}) \)
\( Q1 := Q \text{ quo } Q1 ; \, d1 := \text{degree}(Q1) \)
\( \text{if } d1 = d \text{ then result := cons}(Q1, \text{result}) \)
\( \quad \text{else listOfFactors := cons}(Q1, \text{listOfFactors}) \)
\( \text{if } \text{empty?}(\text{listOfFactors}) \text{ then leave} \)
\( Q := \text{first}(\text{listOfFactors}) \)
\( \text{return result} \)

\( \text{internalFactorSquareFree}(P : \text{PolK}) : \text{List} (\text{PolK}) == -- P \text{ is monic-squareFree} \)
\( \text{degree}(P) = 1 \Rightarrow [P] \)
\( \text{result} : \text{List} (\text{PolK}) := [] \)
\( \text{Xpowers} : \text{PrimitiveArray} (\text{PolK}) := p\text{Powers}(P) \)
\( S : \text{PolK} := \text{Xpowers}(1) \)
\( \text{for } j \text{ in } 1 \ldots \text{primeKdim}-1 \text{ repeat S := pExp(S, Xpowers)} \)
\( \text{for } i \text{ in } 1 \ldots \text{repeat} \quad -- S = \text{X}**(q**i) \mod P \)
\( \text{if } \text{degree}(R := \text{gcd}(S - X, P)) > 0 \text{ then} \)
result := concat(internalFactorCZ(R, i, Xpowers), result)
if degree (P) = degree (R) then return result
P := P quo R
if i >= degree(P) quo 2 then return cons(P, result)
for j in 0 .. degree(P)-1 repeat Xpowers(j):=Xpowers(j) rem P
S := S rem P
else if i >= degree(P) quo 2 then return cons(P, result)
for j in 1 .. primeKdim repeat S := pExp(S, Xpowers)

internalFactor(P:PolK, sqrfree:PolK -> Factored(PolK)) : Factored(PolK) ==
result : Factored(PolK)
if (d := minimumDegree(P)) > 0 then
P := P quo monomial(1, d)
result := primeFactor(X, d)
else
result := 1
degree(P) = 0 => P * result
if (lcp := leadingCoefficient(P)) ^= 1 then result := inv(lcp) * P
degree(P) = 1 => lcp::PolK * primeFactor(P, 1) * result
sqfP : Factored(PolK) := sqrfree(P)
for x in factors(sqfP) repeat
xFactors : List(PolK) := internalFactorSquareFree(x.factor)
result:= result */[primeFactor(Q, x.exponent) for Q in xFactors]
return lcp::PolK * result

factorUsingYun(P : PolK) : Factored(PolK) == internalFactor(P, Yun)
factorUsingMusser(P : PolK) : Factored(PolK) == internalFactor(P, Musser)
factor(P : PolK) : Factored(PolK) == factorUsingYun(P)
factorSquareFree(P : PolK) : List(PolK) ==
degree(P) = 0 => []
discriminant(P) = 0 => error("factorSquareFree : non quadratfrei")
if (lcp := leadingCoefficient(P)) ^= 1 then P := inv(lcp) * P
return internalFactorSquareFree(F)
factorCantorZassenhaus(P : PolK, d : NonNegativeInteger) : List(PolK) ==
if (lcp := leadingCoefficient(P)) ^= 1 then P := inv(lcp) * P
degree(P) = 1 => [P]
return internalFactorCZ(P, d, pPowers(P))
qExp(Q : PolK, XqPowers : PrimitiveArray(PolK)) : PolK ==
Q' : PolK := 0
while Q ^= 0 repeat
Q' := Q' + leadingCoefficient(Q) * XqPowers(degree(Q))
Q := reductum(Q)
return Q'
qPowers (Xq : PolK, P : PolK) : PrimitiveArray(PolK) == -- Xq = X**q mod P
n := degree(P)
result : PrimitiveArray(PolK) := new(n, 1)
result(1) := Q := Xq
for i in 2 .. n-1 repeat (Q := Q*Xq rem P ; result(i) := Q)
return result

discriminantTest?(P : PolK) : Boolean ==
  (delta : K := discriminant(P)) = 0 => true
  StickelbergerTest : Boolean := (delta ** q’ = 1) = even?(degree(P))
  return StickelbergerTest

evenCharacteristicIrreducible?(P : PolK) : Boolean ==
  (n := degree(P)) = 0 => false
  n = 1 => true
  degree(gcd(P, D(P))) > 0 => false
  if (lcP := leadingCoefficient(P)) = 1 then P := inv(lcP) * P
  S : PolK := exp(X, q, P)
  if degree(gcd(S - X, P)) > 0 then
    return false
  if n < 4 then return true
  maxDegreeToTest : NonNegativeInteger := n quo 2
  XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
  for i in 2 .. maxDegreeToTest repeat
    S := qExp(S, XqPowers)
    if degree(gcd(S - X, P)) > 0 then
      return false
  return true

oddCharacteristicIrreducible?(P : PolK) : Boolean ==
  (n := degree(P)) = 0 => false
  n = 1 => true
  discriminantTest?(P) => false
  if (lcP := leadingCoefficient(P)) = 1 then P := inv(lcP) * P
  S : PolK := exp(X, q, P)
  if degree(gcd(S - X, P)) > 0 then
    return false
  if n < 6 then return true
  maxDegreeToTest : NonNegativeInteger := n quo 3
  XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
  for i in 2 .. maxDegreeToTest repeat
    S := qExp(S, XqPowers)
    if degree(gcd(S - X, P)) > 0 then
      return false
  return true

if p = 2 then
  irreducible?(P : PolK) : Boolean == evenCharacteristicIrreducible?(P)
else
irreducible?(P : PolK) : Boolean == oddCharacteristicIrreducible?(P)

---

FFFACTOR.dotabb

"FFFACTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFACTOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFACTOR" -> "PFECAT"

---

package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect

--- FiniteFieldFactorizationWithSizeParseBySideEffect.input ---

)set break resume
/sys rm -f FiniteFieldFactorizationWithSizeParseBySideEffect.output
/spool FiniteFieldFactorizationWithSizeParseBySideEffect.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldFactorizationWithSizeParseBySideEffect
--R
--R FiniteFieldFactorizationWithSizeParseBySideEffect(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory(K)) is a package constructor
--R Abbreviation for FiniteFieldFactorizationWithSizeParseBySideEffect is FFFACTSE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFACTSE
--R
--R----------------------------------- Operations -----------------------------------
--R factorCantorZassenhaus : (PolK,NonNegativeInteger) -> List(PolK)
--R factorSquareFree : PolK -> List(PolK)
--R factorUsingMusser : PolK -> Factored(PolK)
--R factorUsingYun : PolK -> Factored(PolK)
--R
--E 1

)spool
)lisp (bye)
FiniteFieldFactorizationWithSizeParseBySideEffect examples

Part of the package for Algebraic Function Fields in one variable (PAFF)

It has been modified (very slightly) so that each time the "factor" function is used, the variable related to the size of the field over which the polynomial is factorized is reset. This is done in order to be used with a "dynamic extension field" which size is not fixed but set before calling the "factor" function and which is parse by side effect to this package via the function "size". See the local function "initialize" of this package.

See Also:
o )show FiniteFieldFactorizationWithSizeParseBySideEffect

Exports:
factor factorCantorZassenhaus factorSquareFree
factorUsingMusser factorUsingYun irreducible?

)— package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect —

)abbrev package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect
PACKAGE FFFACTSE FINITEFIELDFACTORIZATIONWITHSIZEPARSEBYSIDEFFECT

++ Author: Patrice Naudin, Claude Quitté, Kaj Laursen
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
++ It has been modified (very slantly) so that each time the "factor"
++ function is used, the variable related to the size of the field
++ over which the polynomial is factorized is reset. This is done in
++ order to be used with a "dynamic extension field" which size is not
++ fixed but set before calling the "factor" function and which is
++ parse by side effect to this package via the function "size". See
++ the local function "initialize" of this package.

FiniteFieldFactorizationWithSizeParseBySideEffect(K : FiniteFieldCategory, PolK : UnivariatePolynomialCategory(K)) : with

    factorSquareFree : PolK -> List(PolK)
    factorCantorZassenhaus : (PolK, NonNegativeInteger) -> List(PolK)
    factor : PolK -> Factored(PolK)
    factorUsingYun : PolK -> Factored(PolK)
    factorUsingMusser : PolK -> Factored(PolK)
    irreducible? : PolK -> Boolean

== add

import FiniteFieldSquareFreeDecomposition(K, PolK)

p : NonNegativeInteger := characteristic()$K
p' : NonNegativeInteger := p quo 2 -- used for odd p : (p-1)/2
q : NonNegativeInteger := size()$K
q' : NonNegativeInteger := q quo 2 -- used for odd q : (q-1)/2
X : PolK := monomial(1, 1)
primeKdim : NonNegativeInteger :=
    q_quo_p : NonNegativeInteger := q quo p ; e : NonNegativeInteger := 1
    while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)
e
initialize(): Void() ==
    q : NonNegativeInteger := size()$K
    q' : NonNegativeInteger := q quo 2 -- used for odd q : (q-1)/2
primeKdim : NonNegativeInteger :=
    q_quo_p : NonNegativeInteger := q quo p ; e:NonNegativeInteger := 1
    while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)
e
exp(P : PolK, n : NonNegativeInteger, R : PolK) : PolK ==
    PP : PolK := P rem R ; Q : PolK := 1
    repeat
        if odd?(n) then Q := Q * PP rem R
        (n := n quo 2) = 0 => leave
        PP := PP * PP rem R
    return Q


pPowers(P : PolK) : PrimitiveArray(PolK) == -- P is monic
  n := degree(P)
  result : PrimitiveArray(PolK) := new(n, 1)
  result(1) := Qi := Q := exp(X, p, P)
  for i in 2 .. n-1 repeat (Qi := Qi*Q rem P ; result(i) := Qi)
  return result

pExp(Q : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
  Q' : PolK := 0
  while Q ^= 0 repeat
    Q' := Q' + primeFrobenius(leadingCoefficient(Q)) * Xpowers(degree(Q))
    Q := reductum(Q)
  return Q'

pTrace(Q : PolK, d : NonNegativeInteger, P : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
  Q : PolK := Q rem P
  result : PolK := Q
  for i in 1 .. d-1 repeat result := Q + pExp(result, Xpowers)
  return result rem P

random(n : NonNegativeInteger) : PolK ==
  repeat
    if (deg := (random(n)$Integer)::NonNegativeInteger) > 0 then leave
  repeat
    if (x : K := random()$K) ^= 0 then leave
  result : PolK :=
    monomial(x, deg) + +/
    [monomial(random()$K, i) for i in 0 .. deg-1]
  return result

internalFactorCZ(P : PolK, -- P monic-squarefree
  d:NonNegativeInteger, Xpowers:PrimitiveArray(PolK)) : List(PolK) ==
  listOfFactors : List(PolK) := [P]
  degree(P) = d => return listOfFactors
  result : List(PolK) := []
  pDim : NonNegativeInteger := d * primeKdim
  Q : PolK := P
  repeat
    G := pTrace(random(degree(Q)), pDim, Q, Xpowers)
    if p > 2 then G := exp(G, p', Q) - 1
    Q1 := gcd(G, Q) ; d1 := degree(Q1)
    if d1 > 0 and d1 < degree(Q) then
      listOfFactors := rest(listOfFactors)
      if d1 = d then result := cons(Q1, result)
      else listOfFactors := cons(Q1, listOfFactors)
    Q1 := Q quo Q1 ; d1 := degree(Q1)
    if d1 = d then result := cons(Q1, result)
else listOfFactors := cons(Q1, listOfFactors)
if empty?(listOfFactors) then leave
Q := first(listOfFactors)
return result

internalFactorSquareFree(P:PolK):List(PolK) == -- P is monic-squareFree
degree(P) = 1 => [P]
result : List(PolK) := []
Xpowers : PrimitiveArray(PolK) := pPowers(P)
S : PolK := Xpowers(1)
for j in 1..primeKdim-1 repeat S := pExp(S, Xpowers)
for i in 1 .. repeat -- S = X**(q**i) mod P
if degree(R := gcd(S - X, P)) > 0 then
result := concat(internalFactorCZ(R, i, Xpowers), result)
if degree (P) = degree (R) then return result
P := P quo R
if i >= degree(P) quo 2 then return cons(P, result)
for j in 0 .. degree(P)-1 repeat Xpowers(j):=Xpowers(j) rem P
else if i >= degree(P) quo 2 then return cons(P, result)
for j in 1 .. primeKdim repeat S := pExp(S, Xpowers)

internalFactor(P:PolK, sqrfree:PolK -> Factored(PolK)) : Factored(PolK) ==
result : Factored(PolK)
if (d := minimumDegree(P)) > 0 then
P := P quo monomial(1, d)
result := primeFactor(X, d)
else
result := 1
degree(P) = 0 => P * result
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => lcP::PolK * primeFactor(P, 1) * result
sqfP : Factored(PolK) := sqrfree(P)
for x in factors(sqfP) repeat
xFactors : List(PolK) := internalFactorSquareFree(x.factor)
result:=result * */[primeFactor(Q, x.exponent) for Q in xFactors]
return lcP::PolK * result

factorUsingYun(P : PolK) : Factored(PolK) == internalFactor(P, Yun)
factorUsingMusser(P : PolK) : Factored(PolK) == internalFactor(P, Musser)
factor(P : PolK) : Factored(PolK) ==
initialize()
factorUsingYun(P)
factorSquareFree(P : PolK) : List(PolK) ==
degree(P) = 0 => []
discriminant(P) = 0 => error("factorSquareFree : non quadratfrei")
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
return internalFactorSquareFree(P)

factorCantorZassenhaus(P : PolK, d : NonNegativeInteger) : List(PolK) ==
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => [P]
return internalFactorCZ(P, d, pPowers(P))

qExp(Q : PolK, XqPowers : PrimitiveArray(PolK)) : PolK ==
Q' : PolK := 0
while Q ^= 0 repeat
  Q' := Q' + leadingCoefficient(Q) * XqPowers(degree(Q))
  Q := reductum(Q)
return Q'

qPowers (Xq:PolK, P:PolK) : PrimitiveArray(PolK) == -- Xq = X**q mod P
  n := degree(P)
  result : PrimitiveArray(PolK) := new(n, 1)
  result(1) := Q := Xq
  for i in 2 .. n-1 repeat (Q := Q*Xq rem P ; result(i) := Q)
return result

discriminantTest?(P : PolK) : Boolean ==
  (delta : K := discriminant(P)) = 0 => true
  StickelbergerTest : Boolean := (delta ** q' = 1) = even?(degree(P))
  return StickelbergerTest

evenCharacteristicIrreducible?(P : PolK) : Boolean ==
  (n := degree(P)) = 0 => false
  n = 1 => true
  degree(gcd(P, D(P))) > 0 => false
  if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
  S : PolK := exp(X, q, P)
  if degree(gcd(S - X, P)) > 0 then
    return false
  if n < 4 then return true
  maxDegreeToTest : NonNegativeInteger := n quo 2
  XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
  for i in 2 .. maxDegreeToTest repeat
    S := qExp(S, XqPowers)
    if degree(gcd(S - X, P)) > 0 then
      return false
  return true

oddCharacteristicIrreducible?(P : PolK) : Boolean ==
  (n := degree(P)) = 0 => false
  n = 1 => true
  discriminantTest?(P) => false
  if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
  S : PolK := exp(X, q, P)
  if degree(gcd(S - X, P)) > 0 then
return false
if n < 6 then return true
maxDegreeToTest : NonNegativeInteger := n quo 3
XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
for i in 2 .. maxDegreeToTest repeat
  S := qExp(S, XqPowers)
  if degree(gcd(S - X, P)) > 0 then
    return false
  return true
if p = 2 then
  irreducible?(P : PolK) : Boolean == evenCharacterIrreducible?(P)
else
  irreducible?(P : PolK) : Boolean == oddCharacterIrreducible?(P)

— FFFACTSE.dotabb —

"FFFACTSE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFACTSE"]
"FFSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFACTSE" -> "FFSQFR"
"FFFACTSE" -> "PFECAT"

— package FFF FiniteFieldFunctions —

)set break resume
)sys rm -f FiniteFieldFunctions.output
)spool FiniteFieldFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FiniteFieldFunctions
--E 1

)spool
)lisp (bye)
FiniteFieldFunctions is a package with functions concerning finite extension fields of the finite ground field GF, e.g. Zech logarithms.

See Also:
- )show FiniteFieldFunctions

---

Exports:
- createLowComplexityNormalBasis
- createLowComplexityTable
- createMultiplicationMatrix
- createMultiplicationTable
- createZechTable
- sizeMultiplication

---

)abbrev package FFF FiniteFieldFunctions
++ Author: J. Grabmeier, A. Scheerhorn
++ Date Created: 21 March 1991
++ Date Last Updated: 31 March 1991
++ References:
++ Lidl, R. & Niederreiter, H., "Finite Fields",
++ Encycl. of Math. 20, Addison-Wesley, 1983
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ FiniteFieldFunctions(GF) is a package with functions
++ concerning finite extension fields of the finite ground field GF,
++ e.g. Zech logarithms.

FiniteFieldFunctions(GF): Exports == Implementation where
GF : FiniteFieldCategory -- the ground field
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
I ==> Integer
SI ==> SingleInteger
SUP ==> SparseUnivariatePolynomial GF
V ==> Vector
M ==> Matrix
L ==> List
OUT ==> OutputForm
SAE ==> SimpleAlgebraicExtension
ARR ==> PrimitiveArray(SI)
TERM ==> Record(value:GF,index:SI)
MM ==> ModMonic(GF,SUP)
PF ==> PrimeField

Exports ==> with

createZechTable: SUP -> ARR
++ createZechTable(f) generates a Zech logarithm table for the cyclic
++ group representation of a extension of the ground field by the
++ primitive polynomial f(x), i.e. \spad{Z(i)},
++ defined by x**Z(i) = 1+x**i is stored at index i.
++ This is needed in particular
++ to perform addition of field elements in finite fields represented
++ in this way. See \spadtype{FFCGP}, \spadtype{FFCGX}.
createMultiplicationTable: SUP -> V L TERM
++ createMultiplicationTable(f) generates a multiplication
++ table for the normal basis of the field extension determined
++ by f. This is needed to perform multiplications
++ between elements represented as coordinate vectors to this basis.
++ See \spadtype{FFNBP}, \spadtype{FFNBX}.
createMultiplicationMatrix: V L TERM -> M GF
++ createMultiplicationMatrix(m) forms the multiplication table
++ m into a matrix over the ground field.
-- only useful for the user to visualise the multiplication table
-- in a nice form
sizeMultiplication: V L TERM -> NNI
++ sizeMultiplication(m) returns the number of entries
++ of the multiplication table m.
-- the time of the multiplication of field elements depends
-- on this size
createLowComplexityTable: PI -> Union(Vector List TERM,"failed")
++ createLowComplexityTable(n) tries to find
++ a low complexity normal basis of degree n over GF
++ and returns its multiplication matrix
++ Fails, if it does not find a low complexity basis
createLowComplexityNormalBasis: PI -> Union(SUP, V L TERM)
++ createLowComplexityNormalBasis(n) tries to find a
++ a low complexity normal basis of degree n over GF
++ and returns its multiplication matrix
++ If no low complexity basis is found it calls
++ \axiomFunFrom{createNormalPoly}{FiniteFieldPolynomialPackage}(n)
++ to produce a normal polynomial of degree n over GF

Implementation == add

createLowComplexityNormalBasis(n) ==
  (u:=createLowComplexityTable(n)) case "failed" =>
  createNormalPoly(n)$FiniteFieldPolynomialPackage(GF)
  u::(V L TERM)

-- try to find a low complexity normal basis multiplication table
-- of the field of extension degree n
-- the algorithm is from:
-- Wassermann A., Konstruktion von Normalbasen,

createLowComplexityTable(n) ==
  q:=size()$GF
  -- this algorithm works only for prime fields
  p:=characteristic()$GF
  -- search of a suitable parameter k
  k:NNI:=0
  for i in 1..n-1 while (k=0) repeat
    if prime?(i*n+1) and not(p = (i*n+1)) then
      primitive?(q::PF(i*n+1))$PF(i*n+1) =>
        a:NNI:=1
        k:=i
        t1:PF(k*n+1):=(q::PF(k*n+1))**n
        gcd(n,a:=discreteLog(q::PF(n*i+1))$PF(n*i+1))$I = 1 =>
          k:=i
          t1:=primitiveElement()$PF(k*n+1)**n
        k = 0 => "failed"
  -- initialize some start values
  multmat:M PF(p):=zero(n,n)
p1:=(k*n+1)
pkn:=q::PF(p1)
t:=t1 pretend PF(p1)
if odd?(k) then
  jt:I:=(n quo 2)+1
  vt:I:=positiveRemainder((k-a) quo 2,k)+1
else
  jt:=1
  vt:=(k quo 2)+1
-- compute matrix
vec:=vector(1..k)
for x in 1..k repeat
  for i in 1..n repeat
    vec.((t**(x-1) * pkn**(i-1)) pretend Integer+1):=
      positiveRemainder(l,p1)

lvj:=vector(1..k)
for v in 1..k repeat
  for j in 1..n repeat
    if (j<>jt) or (v<>vt) then
      help:=t**(v-1)*pkn**(j-1)+1@PF(p1)
      setelt(lvj,v,j,vec.(help pretend I +1))
  for j in 1..n repeat
    if j<>jt then
      for v in 1..k repeat
        lvjh:=elt(lvj,v,j)
        setelt(multmat,j,lvjh,elt(multmat,j,lvjh)+1)
  for i in 1..n repeat
    setelt(multmat,jt,i,positiveRemainder(-k,p)::PF(p))

-- multmat
m:=nrows(multmat)$M PF(p)
multtable:=vector(list(1..m))
for i in 1..m repeat
  l:=null
  v:=row(multmat,i)
  for j in (1..n) repeat
    if (v.j <> 0) then
      term:=(-v.j)@I::GF,(j-2) pretend SI$TERM
      l:=cons(term,l)$L TERM
  qsetelt_!(multtable,i,copy l)$V L TERM
multtable

sizeMultiplication(m) ==
s:=0
for i in 1..#m repeat
  s := s + #(m[i])
s

createMultiplicationTable(f:SUP) ==
  sizeGF:=size()$GF -- the size of the ground field
  m:=degree(f)$SUP pretend PI
  m=1 =>
m1 := m - 1
-- initialize basis change matrices
setPoly(f)$MM

e := reduce(monomial(1, 1)$SUP)$MM ** sizeGF

w := 1$MM

qpow : PrimitiveArray(MM) := new(m, 0)
qpow.0 := 1$MM

for i in 1..m1 repeat
  qpow.i := (w := w * e)

qexp := PrimitiveArray(MM) := new(m, 0)
qexp.0 := reduce(monomial(1, 1)$SUP)$MM

mat := zero(m, m)$(M GF)
qsetelt_!(mat, 2, 1, 1$GF)$(M GF)
h := qpow.1
qexp.1 := h

setColumn_!(mat, 2, Vectorise(h)$MM)$(M GF)

for i in 2..m1 repeat
  g := 0$MM
  while h <> 0 repeat
    g := g + leadingCoefficient(h) * qpow.degree(h)$MM
    h := reductum(h)$MM
    qexp.i := g
  setColumn_!(mat, i + 1, Vectorise(h := g)$MM)$(M GF)
-- loop invariant: qexp.i = x**(q**i)

mat1 := inverse(mat)$(M GF)
mat1 = "failed" =>
  error "createMultiplicationTable: polynomial must be normal"

mat := mat1 :: (M GF)
-- initialize multiplication table

multtable := V L TERM := new(m, nil()$(L TERM))$(V L TERM)

for i in 1..m repeat
  l : L TERM := nil()$(L TERM)
  v : V GF := mat *$(M GF) Vectorise(qexp.(i-1) *$MM qexp.0)$MM
  for j in (1::SI)..(m::SI) repeat
    if (v.j <> 0$GF) then
      term := TERM := [(v.j), j-(2::SI)]$TERM
      l := cons(term, l) $(L TERM)
  qsetelt_!(multtable, i, copy l)$/(V L TERM)
multtable

createZechTable(f:$SUP) ==
  sizeGF := size()$GF -- the size of the ground field
  m := degree(f)$SUP::PI
  qn1 := (sizeGF ** m - 1) pretend SI
  zechlog := ARR := new(((sizeGF ** m + 1) quo 2)::NNI, -1::SI)$ARR
  helparr := ARR := new(sizeGF ** m :: NNI, 0$SI)$ARR
  primElement := reduce(monomial(1, 1)$SUP)$SAE(GF, SUP, f)
a:=primElement
for i in 1..qm1-1 repeat
    helparr.(lookup(a -$SAE(GF,SUP,f)$ 1$SAE(GF,SUP,f)$ _
    )$SAE(GF,SUP,f)):=i::SI
a:=a * primElement
characteristic() = 2 =>
a:=primElement
for i in 1..(qm1 quo 2) repeat
    zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
a:=a * primElement
zechlog
a:=1$SAE(GF,SUP,f)
for i in 0..((qm1-2) quo 2) repeat
    zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
a:=a * primElement
zechlog

createMultiplicationMatrix(m) ==
n:NNI:=#m
mat:M GF:=zero(n,n)$(M GF)
for i in 1..n repeat
    for t in m.i repeat
        qsetelt_!(mat,i,t.index+2,t.value)
mat

---

--- FFF.dotabb ---

"FFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFF" -> "PFECAT"

---

package FFHOM FiniteFieldHomomorphisms

--- FiniteFieldHomomorphisms.input ---

)set break resume
)sys rm -f FiniteFieldHomomorphisms.output
)spool FiniteFieldHomomorphisms.output
)set message test on
)set message auto off
FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of elements between the fields F1 and F2, which both must be finite simple algebraic extensions of the finite ground field GF.

See Also:
- )show FiniteFieldHomomorphisms

Exports:
- coerce

---

package FFHOM FiniteFieldHomomorphisms ---
PACKAGE FFHOM FiniteFieldHomomorphisms
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopaedia of Mathematics and
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of
++ elements between the fields F1 and F2, which both must be
++ finite simple algebraic extensions of the finite ground field GF.

FiniteFieldHomomorphisms(F1,GF,F2): Exports == Implementation where
  F1: FiniteAlgebraicExtensionField(GF)
  GF: FiniteFieldCategory
  F2: FiniteAlgebraicExtensionField(GF)
  -- the homorphism can only convert elements w.r.t. the last extension .
  -- Adding a function 'groundField()' which returns the groundfield of GF
  -- as a variable of type FiniteFieldCategory in the new compiler, one
  -- could build up 'convert' recursively to get an homomorphism w.r.t
  -- the whole extension.
  I ==> Integer
  NNI ==> NonNegativeInteger
  SI ==> SingleInteger
  PI ==> PositiveInteger
  SUP ==> SparseUnivariatePolynomial
  M ==> Matrix GF
  FFP ==> FiniteFieldExtensionByPolynomial
  FFPOL2 ==> FiniteFieldPolynomialPackage2
  FFPOLY ==> FiniteFieldPolynomialPackage
  OUT ==> OutputForm

Exports == with

  coerce: F1 -> F2
  ++ coerce(x) is the homomorphic image of x from
  ++ F1 in F2. Thus coerce is a
  ++ field homomorphism between the fields extensions
  ++ F1 and F2 both over ground field GF
  ++ (the second argument to the package).
  ++ Error: if the extension degree of F1 doesn't divide
  ++ the extension degree of F2.
  ++ Note that the other coercion function in the
  ++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.

  coerce: F2 -> F1
  ++ coerce(x) is the homomorphic image of x from
  ++ F2 in F1, where coerce is a
++ field homomorphism between the fields extensions
++ F2 and F1 both over ground field GF
++ (the second argument to the package).
++ Error: if the extension degree of F2 doesn't divide
++ the extension degree of F1.
++ Note that the other coercion function in the
++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.
-- coerce(coerce(x:F1)@F2)@F1 = x and coerce(coerce(y:F2)@F1)@F2 = y

Implementation ==> add

-- global variables ===================================================

degree1:NNI:= extensionDegree()$F1
degree2:NNI:= extensionDegree()$F2
-- the degrees of the last extension

-- a necessary condition for the one field being an subfield of
-- the other one is, that the respective extension degrees are
-- multiples
if max(degree1,degree2) rem min(degree1,degree2) ^= 0 then
error "FFHOM: one extension degree must divide the other one"

conMat1to2:M:= zero(degree2,degree1)$M
-- conversion Matix for the conversion direction F1 -> F2
conMat2to1:M:= zero(degree1,degree2)$M
-- conversion Matix for the conversion direction F2 -> F1

repType1:=representationType()$F1
repType2:=representationType()$F2
-- the representation types of the fields

init?:Boolean:=true
-- gets false after initialization

defPol1:=definingPolynomial()$F1
defPol2:=definingPolynomial()$F2
-- the defining polynomials of the fields

-- functions ===========================================================

compare: (SUP GF,SUP GF) -> Boolean
-- compares two polynomials

convertWRTsameDefPol12: F1 -> F2
convertWRTsameDefPol21: F2 -> F1
-- homomorphism if the last extension of F1 and F2 was build up
-- using the same defining polynomials
convertWRTdifferentDefPol12: F1 -> F2
convertWRTdifferentDefPol21: F2 -> F1
-- homomorphism if the last extension of F1 and F2 was built up
-- with different defining polynomials

initialize: () -> Void
-- computes the conversion matrices

compare(g:(SUP GF),f:(SUP GF)) ==
  degree(f)$(SUP GF) >$NNI degree(g)$(SUP GF) => true
  degree(f)$(SUP GF) <$NNI degree(g)$(SUP GF) => false
  equal:Integer:=0
  for i in degree(f)$(SUP GF)..0 by -1 while equal=0 repeat
    not zero?(coefficient(f,i)$(SUP GF))$GF and _
      zero?(coefficient(g,i)$(SUP GF))$GF => equal:=1
    not zero?(coefficient(g,i)$(SUP GF))$GF and _
      zero?(coefficient(f,i)$(SUP GF))$GF => equal:=-1
    (f1:=lookup(coefficient(f,i)$(SUP GF))$GF) >$PositiveInteger _
    (g1:=lookup(coefficient(g,i)$(SUP GF))$GF) => equal:=1
    f1 <$PositiveInteger g1 => equal:=-1
  equal=1 => true
  false

initialize() ==
-- 1) in the case of equal def. polynomials initialize is called only
-- if one of the rep. types is "normal" and the other one is "polynomial"
-- we have to compute the basis change matrix 'mat', which i-th
-- column are the coordinates of a**(q**i), the i-th component of
-- the normal basis ('a' the root of the def. polynomial and q the
-- size of the groundfield)
defPol1 =$GF defPol2 =>
-- new code using reducedQPowers
mat:=zero(degree1,degree1)$M
arr:=reducedQPowers(defPol1)$FFPOLY(GF)
for i in 1..degree1 repeat
  setColumn_!(mat,i,vectorise(arr.(i-1),degree1)$SUP(GF))$M
-- old code
-- here one of the representation types must be "normal"
-- a:=basis()$FFP(GF,defPol1).2 -- the root of the def. polynomial
-- setColumn_!(mat,1,coordinates(a)$FFP(GF,defPol1))$M
-- for i in 2..degree1 repeat
--  a:= a **$FFP(GF,defPol1) size()$GF
--  setColumn_!(mat,i,coordinates(a)$FFP(GF,defPol1))$M
-- for the direction "normal" -> "polynomial" we have to multiply the
-- coordinate vector of an element of the normal basis field with
-- the matrix 'mat'. In this case 'mat' is the correct conversion
-- matrix for the conversion of F1 to F2, its inverse the correct
-- inversion matrix for the conversion of F2 to F1
repType1 = "normal" => -- repType2 = "polynomial"
conMat1to2:=copy(mat)
conMat2to1:=copy(inverse(mat))$M :: M
-- we finish the function for one case, hence reset initialization flag
init? := false
void()$Void
-- print("'normal' <=> 'polynomial' matrices initialized":OUT)
-- in the other case we have to change the matrices
-- repType2 = "normal" and repType1 = "polynomial"
conMat2to1:=copy(mat)
conMat1to2:=copy(inverse(mat))$M :: M
-- print("'normal' <=> 'polynomial' matrices initialized":OUT)
-- we finish the function for one case, hence reset initialization flag
init? := false
void()$Void
-- 2) in the case of different def. polynomials we have to order the
-- fields to get the same isomorphism, if the package is called with
-- the fields F1 and F2 swapped.
dPbig:= defPol2
rTbig:= repType2
dPsmall:= defPol1
rTsmall:= repType1
degbig:=degree2
degsmall:=degree1
if compare(defPol2,defPol1) then
  degsmall:=degree2
degbig:=degree1
  dPbig:= defPol1
  rTbig:= repType1
dPsmall:= defPol2
  rTsmall:= repType2
-- 3) in every case we need a conversion between the polynomial
-- represented fields. Therefore we compute 'root' as a root of the
-- 'smaller' def. polynomial in the 'bigger' field.
-- We compute the matrix 'matsb', which i-th column are the coordinates
-- of the (i-1)-th power of root, i=1..degsmall. Multiplying a
-- coordinate vector of an element of the 'smaller' field by this
-- matrix, we got the coordinates of the corresponding element in the
-- 'bigger' field.
-- compute the root of dPsmall in the 'big' field
root:=rootOfIrreduciblePoly(dPsmall)$FFPOL2(FFP(GF,dPbig),GF)
-- set up matrix for polynomial conversion
matsb:=zero(degbig,degsmall)$M
qsetelt_!(matsb,1,1,1$GF)$M
a:=root
for i in 2..degsmall repeat
  setColumn_!(matsb,i,coordinates(a)$FFP(GF,dPbig))$M
  a := a *$FFP(GF,dPbig) root
-- the conversion from 'big' to 'small': we can't invert matsb
-- directly, because it has degbig rows and degsmall columns and
-- may be no square matrix. Therefore we construct a square matrix
-- mat from degsmall linear independent rows of matsb and invert it.
-- Now we get the conversion matrix 'matsb' for the conversion from
-- 'big' to 'small' by putting the columns of mat at the indices
-- of the linear independent rows of matsb to columns of matsb.
ra:I:=1 -- the rank
mat:M:=transpose(row(matsb,1))$M -- has already rank 1
rowind:I:=2
iVec:Vector I:=new(degsmall,1$I)$($Vector I)
while ra < degsmall repeat
  if rank(vertConcat(mat,transpose(row(matsb,rowind))$M)$M)$M > ra then
    mat:=vertConcat(mat,transpose(row(matsb,rowind))$M)$M
    ra:=ra+1
    iVec.ra := rowind
    rowind:=rowind + 1
mat:=inverse(mat)$M :: M
matsb:=zero(degsmall,degbig)$M
for i in 1..degsmall repeat
  setColumn_!(matsb,iVec.i,column(mat,i)$M)$M
-- print(matsb::OUT)
-- print(matbs::OUT)
-- 4) if the 'bigger' field is "normal" we have to compose the
-- polynomial conversion with a conversion from polynomial to normal
-- between the FFP(GF,dPbig) and FFNBP(GF,dPbig) the 'bigger'
-- field. Therefore we compute a conversion matrix 'mat' as in 1)
-- Multiplying with the inverse of 'mat' yields the desired
-- conversion from polynomial to normal. Multiplying this matrix by
-- the above computed 'matsb' we got the matrix for converting form
-- 'small polynomial' to 'big normal'.
-- set up matrix 'mat' for polynomial to normal
if rTbig = "normal" then
  arr:=reducedQPowers(dPbig)$FFPOLY(GF)
  mat:=zero(degbig,degbig)$M
  for i in 1..degbig repeat
    setColumn_!(mat,i,vectorise(arr.(i-1),degbig)$SUP(GF))$M
-- old code
--a:=basis()$FFP(GF,dPbig).2 -- the root of the def.Polynomial
--setColumn_!(mat,1,coordinates(a)$FFP(GF,dPbig))$M
--for i in 2..degbig repeat
-- a:= a **$FFP(GF,dPbig) size()$GF
-- setColumn_!(mat,i,coordinates(a)$FFP(GF,dPbig))$M
-- print(inverse(mat)$M::OUT)
  matsb:= (inverse(mat)$M :: M) * matsb
-- print("inv *..":OUT)
matbs:=matbs * mat
-- 5) if the 'smaller' field is "normal" we have first to convert
-- from 'small normal' to 'small polynomial', that is from
-- FFNBP(GF,dPsmall) to FFP(GF,dPsmall). Therefore we compute a
-- conversion matrix 'mat' as in 1). Multiplying with 'mat'
-- yields the desired conversion from normal to polynomial.
-- Multiplying the above computed 'matsb' with 'mat' we got the
-- matrix for converting form 'small normal' to 'big normal'.
-- set up matrix 'mat' for normal to polynomial
if rTsmall = "normal" then
  arr:=reducedQPowers(dPsmall)$FFPOLY(GF)
  mat:=zero(degsmall,degsmall)$M
  for i in 1..degsmall repeat
    setColumn_!(mat,i,vectorise(arr.(i-1),degsmall)$SUP(GF))$M
-- old code
--b:FFP(GF,dPsmall):=basis()$FFP(GF,dPsmall).2
--setColumn_!(mat,1,coordinates(b)$FFP(GF,dPsmall))$M
--for i in 2..degsmall repeat
-- b:= b **$FFP(GF,dPsmall) size()$GF
-- setColumn_!(mat,i,coordinates(b)$FFP(GF,dPsmall))$M
-- print(mat::OUT)
matsb:= matsb * mat
matbs:= (inverse(mat) :: M) * matbs
-- now 'matsb' is the corret conversion matrix for 'small' to 'big'
-- and 'matbs' the corret one for 'big' to 'small'.
-- depending on the above ordering the conversion matrices are
-- initialized

dPbig =$(SUP GF) defPol2 =>$
  conMat1to2 :=matsb
  conMat2to1 :=matbs
  -- print(conMat1to2::OUT)
  -- print(conMat2to1::OUT)
  -- print("conversion matrices initialized":::OUT)
  --we finish the function for one case, hence reset initialization flag
  init? := false
void()$Void

coerce(x:F1) ==
  inGroundField?(x)$F1 => retract(x)$F1 :: F2
  -- if x is already in GF then we can use a simple coercion
  defPol1 =$$(SUP GF) defPol2 =>$ convertWRTsameDefPol12(x)
  convertWRTdifferentDefPol12(x)

convertWRTsameDefPol12(x:F1) ==
  repType1 = repType2 =>$ x pretend F2
  -- same groundfields, same defining polynomials, same
  -- representation types --> F1 = F2, x is already in F2
  repType1 = "cyclic" =>
\[
x = 0_{F1} \Rightarrow 0_{F2}
\]
-- the SI corresponding to the cyclic representation is the exponent of
-- the primitiveElement, therefore we exponentiate the primitiveElement
-- of F2 by it.
primitiveElement()_{F2}^{**_{F2}} (x pretend SI)
repType2 = "cyclic" =>
x = 0_{F1} \Rightarrow 0_{F2}
-- to get the exponent, we have to take the discrete logarithm of the
-- element in the given field.
(discreteLog(x)_{F1} pretend SI) pretend F2
repType2 = "cyclic" =>
-- here one of the representation types is "normal"
if init? then initialize()
-- here a conversion matrix is necessary, (see initialize())
represents(conMat2to1 *$(Matrix GF) coordinates(x)_{F1})_{F2}

convertWRTdifferentDefPol12(x:F1) ==
if init? then initialize()
-- if we want to convert into a 'smaller' field, we have to test,
-- whether the element is in the subfield of the 'bigger' field, which
-- corresponds to the 'smaller' field
if degree1 > degree2 then
  if positiveRemainder(degree2,degree(x)_{F1}) = 0 then
    error "coerce: element doesn't belong to smaller field"
represents(conMat1to2 *$(Matrix GF) coordinates(x)_{F1})_{F2}

-- the three functions below equal the three functions above up to
-- '1' exchanged by '2' in all domain and variable names

coerce(x:F2) ==
inGroundField?(x)_{F2} \Rightarrow retract(x)_{F2} :: F1
-- if x is already in GF then we can use a simple coercion
defPol1 =$\langle SUP GF \rangle defPol2 => convertWRTsameDefPol21(x)
convertWRTdifferentDefPol21(x)

convertWRTsameDefPol21(x:F2) ==
repType1 = repType2 \Rightarrow x pretend F1
-- same groundfields, same defining polynomials,
-- same representation types --> F1 = F2, that is:
-- x is already in F1
repType2 = "cyclic" =>
x = 0_{F2} \Rightarrow 0_{F1}
primitiveElement()_{F1}^{**_{F1}} (x pretend SI)
repType1 = "cyclic" =>
x = 0_{F2} \Rightarrow 0_{F1}
(discreteLog(x)_{F2} pretend SI) pretend F1
-- here one of the representation types is "normal"
if init? then initialize()
represents(conMat2to1 *$(Matrix GF) coordinates(x)_{F2})_{F1}
convertWRTdifferentDefPol21(x:F2) ==
if init? then initialize()
if degree2 > degree1 then
  if positiveRemainder(degree1,degree(x)$F2)^= 0 then
    error "coerce: element doesn't belong to smaller field"
  represents(conMat2to1 *$(Matrix GF) coordinates(x)$F2)$F1

package FFPOLY FiniteFieldPolynomialPackage

FiniteFieldPolynomialPackage.help

====================================================================
FiniteFieldPolynomialPackage examples
====================================================================

--- S 1 of 1
)show FiniteFieldPolynomialPackage
-- E 1

)spool
)lisp (bye)
This package provides a number of functions for generating, counting and testing irreducible, normal, primitive, random polynomials over finite fields.

See Also:
o )show FiniteFieldPolynomialPackage

---

**FiniteFieldPolynomialPackage (FFPOLY)**

**Exports:**
- createIrreduciblePoly
- createNormalPoly
- createNormalPrimitivePoly
- createPrimitiveNormalPoly
- nextIrreduciblePoly
- nextNormalPoly
- nextNormalPrimitivePoly
- nextPrimitiveNormalPoly
- nextPrimitivePoly
- numberOfIrreduciblePoly
- numberOfNormalPoly
- numberOfPrimitivePoly
- primitive?
- leastAffineMultiple
- normal?
- random
- reducedQPowers

---

)abbrev package FFPOLY FiniteFieldPolynomialPackage
++ Author: A. Bouyer, J. Grabmeier, A. Scheerhorn, R. Sutor, B. Trager
++ Date Created: January 1991
++ Date Last Updated: 1 June 1994
++ References:
++ J. Grabmeier, A. Scheerhorn: Finite Fields in Axiom.
++ Description:
This package provides a number of functions for generating, counting and testing irreducible, normal, primitive, random polynomials over finite fields.

FiniteFieldPolynomialPackage GF : Exports == Implementation where

GF : FiniteFieldCategory

I    ===> Integer
L    ===> List
NNI ===> NonNegativeInteger
PI   ===> PositiveInteger
Rec  ===> Record(expnt:NNI, coeff:GF)
Repr ===> L Rec
SUP  ===> SparseUnivariatePolynomial GF

Exports ==> with

-- qEulerPhiCyclotomic : PI -> PI
-- ++ qEulerPhiCyclotomic(n)$FFPOLY(GF) yields the q-Euler's function
-- ++ of the n-th cyclotomic polynomial over the field GF of
-- ++ order q (cf. [LN] p.122);
-- ++ error if n is a multiple of the field characteristic.

primitive? : SUP -> Boolean
++ primitive?(f) tests whether the polynomial f over a finite field is primitive, i.e. all its roots are primitive.

normal? : SUP -> Boolean
++ normal?(f) tests whether the polynomial f over a finite field is normal, i.e. its roots are linearly independent over the field.

numberOfIrreduciblePoly : PI -> PI
++ numberOfIrreduciblePoly(n)$FFPOLY(GF) yields the number of monic irreducible univariate polynomials of degree n over the finite field GF.

numberOfPrimitivePoly : PI -> PI
++ numberOfPrimitivePoly(n)$FFPOLY(GF) yields the number of primitive polynomials of degree n over the finite field GF.

numberOfNormalPoly : PI -> PI
++ numberOfNormalPoly(n)$FFPOLY(GF) yields the number of normal polynomials of degree n over the finite field GF.

createIrreduciblePoly : PI -> SUP
++ createIrreduciblePoly(n)$FFPOLY(GF) generates a monic irreducible univariate polynomial of degree n over the finite field GF.

createPrimitivePoly : PI -> SUP
++ createPrimitivePoly(n)$FFPOLY(GF) generates a primitive polynomial of degree n over the finite field GF.

createNormalPoly : PI -> SUP
++ createNormalPoly(n)$FFPOLY(GF) generates a normal polynomial of degree n over the finite field GF.

createNormalPrimitivePoly : PI -> SUP
++ createNormalPrimitivePoly(n)$FFPOLY(GF) generates a normal and primitive polynomial of degree n over the field GF.
Note that this function is equivalent to createPrimitiveNormalPoly(n).

createPrimitiveNormalPoly : PI -> SUP
++ createPrimitiveNormalPoly(n)\$FFPOLY(GF) generates a normal and
++ primitive polynomial of degree n over the field GF.
++ polynomial of degree n over the field GF.
nextIrreduciblePoly : SUP -> Union(SUP, "failed")
++ nextIrreduciblePoly(f) yields the next monic irreducible polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if
++ the number of monomials of f is less
++ than this number for g.
++ If f and g have the same number of monomials,
++ the lists of exponents are compared lexicographically.
++ If these lists are also equal, the lists of coefficients
++ are compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
nextPrimitivePoly : SUP -> Union(SUP, "failed")
++ nextPrimitivePoly(f) yields the next primitive polynomial over
++ a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the constant term
++ of f is less than
++ this number for g.
++ If these values are equal, then \spad{f < g} if
++ if the number of monomials of f is less than that for g or if
++ the lists of exponents of f are lexicographically less than the
++ corresponding list for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
nextNormalPoly : SUP -> Union(SUP, "failed")
++ nextNormalPoly(f) yields the next normal polynomial over
++ a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the coefficient
++ of the term of degree
++ n-1 of f is less than that for g.
++ In case these numbers are equal, \spad{f < g} if
++ if the number of monomials of f is less than that for g or if
++ the list of exponents of f are lexicographically less than the
++ corresponding list for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.

nextNormalPrimitivePoly : SUP -> Union(SUP, "failed")
++ nextNormalPrimitivePoly(f) yields the next normal primitive polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the constant
++ term of f is less than
++ this number for g or if
++ lookup of the coefficient of the term of degree n-1
++ of f is less than this number for g.
++ Otherwise, \spad{f < g}
++ if the number of monomials of f is less than
++ that for g or if the lists of exponents for f are
++ lexicographically less than those for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
++ This operation is equivalent to nextPrimitiveNormalPoly(f).

nextPrimitiveNormalPoly : SUP -> Union(SUP, "failed")
++ nextPrimitiveNormalPoly(f) yields the next primitive normal polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the
++ constant term of f is less than
++ this number for g, or, in case these numbers are equal, if the
++ lookup of the coefficient of the term of degree n-1
++ of f is less than this number for g.
++ If these numbers are equals, \spad{f < g}
++ if the number of monomials of f is less than
++ that for g, or if the lists of exponents for f are lexicographically
++ less than those for g.
++ If these lists are also equal, the lists of coefficients are
++ coefficients according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
++ This operation is equivalent to nextNormalPrimitivePoly(f).

-- random : () -> SUP
-- ++ random()$FFPOLY(GF) generates a random monic polynomial
-- ++ of random degree over the field GF

random : PI -> SUP
++ random(n)$FFPOLY(GF) generates a random monic polynomial
++ of degree n over the finite field GF.

random : (PI, PI) -> SUP
++ random(m,n)$FFPOLY(GF) generates a random monic polynomial
++ of degree d over the finite field GF, d between m and n.

leastAffineMultiple : SUP -> SUP
++ leastAffineMultiple(f) computes the least affine polynomial which
++ is divisible by the polynomial \( f \) over the finite field \( GF \),
++ i.e. a polynomial whose exponents are 0 or a power of \( q \), the
++ size of \( GF \).

\textbf{reducedQPowers:} \( \text{SUP} \rightarrow \text{PrimitiveArray SUP} \)
++ \text{reducedQPowers}(f)
++ generates \( \text{\spad{[x, x**q, x**(q**2), \ldots, x**(q**(n-1))]} } \)
++ reduced modulo \( f \) where \( \text{\spad{q = size()}$GF} \) and \( \text{\spad{n = degree f}} \).

\[ \text{-- we intend to implement also the functions} \]
\[ \text{-- cyclotomicPoly: PI -> SUP, order: SUP -> PI,} \]
\[ \text{-- and maybe a new version of irreducible?} \]

\textbf{Implementation} \( \Rightarrow \) add

\textbf{import IntegerNumberTheoryFunctions}
\textbf{import DistinctDegreeFactorize(GF, SUP)}

\textbf{MM := ModMonic(GF, SUP)}

\textbf{sizeGF : PI := size()}$GF :: PI}

\textbf{revListToSUP(l:Repr):SUP ==}
\textbf{newl:Repr := empty()}
\textbf{-- cannot use map since copy for Record is an XLAM}
\textbf{for t in l repeat newl := cons(copy t, newl)}
\textbf{newl pretend SUP}

\textbf{listToSUP(l:Repr):SUP ==}
\textbf{newl:Repr := [copy t for t in l]}
\textbf{newl pretend SUP}

\textbf{nextSubset : (L NNI, NNI) \rightarrow \text{Union(L NNI, "failed")}}
\textbf{-- for a list \( s \) of length \( m \) with \( 1 \leq s.1 < \ldots < s.m \leq \text{bound} \),}
\textbf{-- \text{nextSubset}(s, \text{bound}) yields the immediate successor of \( s \)}
\textbf{-- (resp. "failed" if \( s = [1, \ldots, \text{bound}] \))}
\textbf{-- where \( s < t \) if and only if:}
\textbf{-- (i) \#s < \#t; or}
\textbf{-- (ii) \#s = \#t and \( s < t \) in the lexicographical order;}
\textbf{-- (we have chosen to fix the signature with NNI instead of PI}
\textbf{-- to avoid coercions in the main functions)}

\textbf{reducedQPowers(f) ==}
\textbf{m:PI:=degree(f)$SUP pretend PI}
\textbf{m1:I:=m-1}
\textbf{setPoly(f)$MM}
\textbf{e:=reduce(monomial(1,1)$SUP)$MM \text{** sizeGF}}
\textbf{w:=1$MM}
\textbf{qpow:PrimitiveArray SUP:=new(m,0)}
```plaintext
for i in 1..m repeat
    qpow.i:=lift(w=w*e)
qexp:=PrimitiveArray SUP:=new(m,0)
m = 1 =>
    qexp.(0$I):= (-coefficient(f,0$NNI)$SUP)::SUP
qexp
qexp.0$I:=monomial(1,1)$SUP
h:=qpow.1
qexp.1:=h
for i in 2..m repeat
    g:=0
    while h ^= 0 repeat
        g:=g + leadingCoefficient(h) * qpow.degree(h)
        h:=reductum(h)
    qexp.i:=(h:=g)
qexp

leastAffineMultiple(f) ==
    -- [LS] p.112
    qexp:=reducedQPowers(f)
    n:=degree(f)$SUP
    b:Matrix GF:= transpose matrix [entries vectorise
                                                                 (qexp.i,n) for i in 0..n-1]
    col1:Matrix GF:= new(n,1,0)
    col1(1,1) := 1
    ns : List Vector GF := nullSpace (horizConcat(col1,b) )
    -- perhaps one should use that the first vector in ns is already
    -- the right one
    ---------------------------------------------------------------
    dim:=n+2
    coeffVector : Vector GF
    until empty? ns repeat
        newCoeffVector := ns.1
        i : PI :=(n+1) pretend PI
        while newCoeffVector(i) = 0 repeat
            i := (i - 1) pretend PI
        if i < dim then
            dim := i
        coeffVector := newCoeffVector
        ns := rest ns
        (coeffVector(1)::SUP) +(+/[monomial(coeffVector.k, _
                                sizeGF**((k-2)::NNI))$SUP for k in 2..dim])

qEulerPhiCyclotomic n ==
    -- n = 1 => (sizeGF - 1) pretend PI
    -- p : PI := characteristic()$GF :: PI
    -- (n rem p) = 0 => error
    "cyclotomic polynomial not defined for this argument value"
    -- q : PI := sizeGF
```

PACKAGE FFPOLY FINITEFIELDPOLYNOMIALPACKAGE

-- determine the multiplicative order of q modulo n
-- e : PI := 1
-- qe : PI := q
-- while (qe rem n) ^= 1 repeat
-- e := e + 1
-- qe := qe * q
-- ((qe - 1) ** ((eulerPhi(n) quo e) pretend PI) ) pretend PI

numberOfIrreduciblePoly n ==
-- we compute the number Nq(n) of monic irreducible polynomials
-- of degree n over the field GF of order q by the formula
-- Nq(n) = (1/n) * sum(moebiusMu(n/d)*q**(d)) where the sum extends
-- over all divisors d of n (cf. [LN] p.93, Th. 3.25)
n := 1 => sizeGF
-- the contribution of d = 1 :
lstd := PI := 1
qd : PI := sizeGF
sum : I := moebiusMu(n) * qd
-- the divisors d > 1 of n :
divisorsOfn : L PI := rest(divisors n) pretend L PI
for d in divisorsOfn repeat
qd := qd * (sizeGF) ** ((d - lstd) pretend PI)
sum := sum + moebiusMu(n quo d) * qd
lstd := d
(sum quo n) :: PI

numberOfPrimitivePoly n == (eulerPhi((sizeGF ** n) - 1) quo n) :: PI
-- [each root of a primitive polynomial of degree n over a field
-- with q elements is a generator of the multiplicative group
-- of a field of order q**n (definition), and the number of such
-- generators is precisely eulerPhi(q**n - 1)]

numberOfNormalPoly n ==
-- we compute the number Nq(n) of normal polynomials of degree n
-- in GF[X], with GF of order q, by the formula
-- Nq(n) = (1/n) * qPhi(X**n - 1) (cf. [LN] p.124) where,
-- for any polynomial f in GF[X] of positive degree n,
-- qPhi(f) = q**n * (1 - q**(n-1)) *...* (1 - q**(n-r)) =
-- q**n * (((q**(n1)-1) / q**(n1)) *...* ((q**(nr)-1) / q**(n_r))),
-- the ni being the degrees of the distinct irreducible factors
-- of f in its canonical factorization over GF
-- hence, if n = m * p**r where p is the characteristic of GF
-- and gcd(m,p) = 1, we get
-- Nq(n) = (1/n)* q**(n-m) * qPhi(X**m - 1)
-- now X**m - 1 is the product of the (pairwise relatively prime)
-- cyclotomic polynomials Qd(X) for which d divides m
-- ([LN] p.64, Th. 2.45), and each Qd(X) factors into
-- eulerPhi(d)/e (distinct) monic irreducible polynomials in GF[X]
-- of the same degree e, where e is the least positive integer k
such that d divides $q^k - 1$ ([LN] p.65, Th. 2.47)
n = 1 => (sizeGF - 1) :: NNI :: PI
m : PI := n
p : PI := characteristic()$GF :: PI
q : PI := sizeGF
while (m rem p) = 0 repeat -- find m such that
  m := (m quo p) :: PI -- n = m * $p^r$ and gcd(m,p) = 1
  m = 1 =>
  -- know that n is a power of p
  $((q ** ((n-1)::NNI) ) * (q - 1) )$ quo n :: PI
prod : I := q - 1
divisorsOfm : L PI := rest(divisors m) pretend L PI
for d in divisorsOfm repeat
  -- determine the multiplicative order of q modulo d
  e : PI := 1
  qe : PI := q
  while (qe rem d) ^= 1 repeat
    e := e + 1
    qe := qe * q
  prod := prod * _
    $((q - 1) ** ((eulerPhi(d) quo e)$ PI)) pretend PI
  (q**((n-m) pretend PI) * prod quo n) pretend PI
primitive? f ==
  -- let GF be a field of order q; a monic polynomial f in GF[X]
  -- of degree n is primitive over GF if and only if its constant
  -- term is non-zero, f divides $X^{(q^n - 1)} - 1$ and,
  -- for each prime divisor d of $q^n - 1$,
  -- f does not divide $X^{((q^n - 1) / d)} - 1$
  -- (cf. [LN] p.89, Th. 3.16, and p.87, following Th. 3.11)
  n : NNI := degree f
  n = 0 => false
  leadingCoefficient f ^= 1 => false
  coefficient(f, 0) = 0 => false
  q : PI := sizeGF
  qn1: PI := (q**n - 1) :: NNI :: PI
  setPoly f
  x := reduce(monomial(1,1)$SUP)$MM -- X rem f represented in MM
  -- may be improved by tabulating the residues $x^{(i*q)}$
  -- for i = 0,...,n-1 :
  --
  lift(x ** qn1)$MM ^= 1 => false -- $X^{(q^n - 1)}$ rem f in GF[X]
lrec : L Record(factor:I, exponent:I) := factors(factor qn1)
lfact : L PI := [] -- collect the prime factors
for rec in lrec repeat -- of $q^n - 1$
  lfact := cons((rec.factor) :: PI, lfact)
for d in lfact repeat
  if (expt := (qn1 quo d)) >= n then
    lift(x ** expt)$MM = 1 => return false
true

normal? f ==
  -- let GF be a field with q elements; a monic irreducible
  -- polynomial f in GF[X] of degree n is normal if its roots
  -- x, x**q, ..., x**(q**(n-1)) are linearly independent over GF
  n := degree f
  n = 0 => false
  leadingCoefficient f ^= 1 => false
  coefficient(f, 0) = 0 => false
  n = 1 => true
  not irreducible? f => false
  g := reducedQPowers(f)
  l := [entries vectorise(g.i, n)$SUP for i in 0..(n-1)::NNI]
  rank(matrix(l)$Matrix(GF)) = n => true
  false

nextSubset(s, bound) ==
  m := #(s)
  m = 0 => [1]
  -- find the first element s(i) of s such that s(i) + 1 < s(i+1) :
  noGap := true
  i := 0
  restOfs := rest s
  while noGap and not empty?(restOfs) repeat
    -- after i steps (0 <= i <= m-1) we have s = [s(i), ..., s(m)]
    -- and restOfs = [s(i+1), ..., s(m)]
    secondOfs := first restOfs -- s(i+1)
    firstOfsPlus1 := first s + 1 -- s(i) + 1
    secondOfs = firstOfsPlus1 =>
      s := restOfs
      i := i + 1
      setfirst_!(s, firstOfsPlus1) -- s := [s(i)+1, s(i+1),..., s(m)]
      noGap := false
    if noGap then -- here s = [s(m)]
      firstOfs := first s
      firstOfs < bound => setfirst_!(s, firstOfs + 1) -- s := [s(m)+1]
      m < bound =>
        setfirst_!(s, m + 1) -- s := [m+1]
        i := m
      return "failed" -- (here m = s(m) = bound)
  for j in 1..1 by -1 repeat -- reconstruct the destroyed
    s := cons(j, s) -- initial part of s
    s

nextIrreduciblePoly f ==
  n := degree f
  n = 0 => error "polynomial must have positive degree"
  -- make f monic
  if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
-- if \( f = f_n X^n + \ldots + f_{i_0} X^{i_0} \) with the \( f_i \) non-zero
-- then \( f_{\text{Repr}} = [n, f_n], \ldots, [i_0, f_{i_0}] \)
\( f_{\text{Repr}} : \text{Repr} := f \text{ pretend Repr} \)
\( \text{fcopy} : \text{Repr} := [] \)
-- we can not simply write \( \text{fcopy} := \text{copy} f_{\text{Repr}} \) because
-- the input(!) \( f \) would be modified by assigning
-- a new value to one of its records
for term in \( f_{\text{Repr}} \) repeat
    \( \text{fcopy} := \text{cons}(\text{copy term}, \text{fcopy}) \)
    if term.expnt ^= 0 then
        \( \text{fcopy} := \text{cons}([0, 0]\$\text{Rec}, \text{fcopy}) \)
    tailpol : \text{Repr} := []
    headpol : \text{Repr} := \text{fcopy} -- \([0, f_0], \ldots, [n, f_n]\) where
        -- \( f_i \) is non-zero for \( i > 0 \)
    \( \text{fcopy} := \text{reverse} \text{fcopy} \)
\( \text{weight} : \text{NNI} := (#(\text{fcopy}) - 1) : : \text{NNI} -- \#s(f) \) as explained above
\( \text{taillookuplist} : \text{L NNI} := [] \)
-- the zeroes in the headlookuplist stand for the \( f_i \)
-- whose lookup’s were not yet computed :
\( \text{headlookuplist} : \text{L NNI} := \text{new}(\text{weight}, 0) \)
\( s : \text{L NNI} := [] -- \text{we will compute} s(f) \) only if necessary
\( n_1 : \text{NNI} := (n - 1) : : \text{NNI} \)
repeat
    -- (run through the possible weights)
while not empty? headlookuplist repeat
    -- find next polynomial in the above order with fixed weight;
    -- assume at this point we have
    -- \( \text{headpol} = [[i_1, f_{i_1}], [i_2, f_{i_2}], \ldots, [n, 1]] \)
    -- and \( \text{tailpol} = [[k, f_k], \ldots, [0, f_0]] \) (with \( k < i_1 \))
    term := first headpol
    \( j := \text{first headlookuplist} \)
if \( j = 0 \) then \( j := \text{lookup(term.coeff)}\$\text{GF} \)
\( j := j + 1 -- \text{lookup}(f(i_1))\$\text{GF} + 1 \)
\( j \text{ rem sizeGF} = 0 => \)
    -- in this case one has to increase \( f_{i_2} \)
\( \text{tailpol} := \text{cons(term, tailpol)} -- [[i_1, f_{i_1}], \ldots, [0, f_0]] \)
\( \text{headpol} := \text{rest headpol} -- [[i_2, f_{i_2}], \ldots, [n, 1]] \)
\( \text{taillookuplist} := \text{cons}(j, \text{taillookuplist}) \)
\( \text{headlookuplist} := \text{rest headlookuplist} \)
-- otherwise set \( f_{i_1} := \text{index}(j)\$\text{GF} \)
\( \text{setelt(first headpol, coeff, index(j :: PI)\$GF) \\text{setfirst_!(headlookuplist, j)} \) \)
if empty? taillookuplist then
    \( \text{pol} := \text{revListToSUP(headpol)} \)
    --
    -- may be improved by excluding reciprocal polynomials
    --
    -- irreducible? \( \text{pol} \) \( \Rightarrow \text{return} \text{pol} \)
else
    -- go back to \( f_k \)
headpol := cons(first tailpol, headpol) -- [[k,fk],...,[n,1]]
tailpol := rest tailpol
headlookuplist := cons(first taillookuplist, headlookuplist)
taillookuplist := rest taillookuplist
-- must search for polynomial with greater weight
if empty? s then -- compute s(f)
    restfcopy := rest fcopy
    for entry in restfcopy repeat s := cons(entry.expnt, s)
weight = n => return "failed"
s1 := nextSubset(rest s, n1) :: L NNI
s := cons(0, s1)
weight := #s
taillookuplist := []
headlookuplist := cons(sizeGF, new((weight-1) :: NNI, 1))
tailpol := []
headpol := [] -- [[0,0], [s.2,1], ... , [s.weight,1], [n,1]] :
s1 := cons(n, reverse s1)
while not empty? s1 repeat
    headpol := cons([first s1, 1]$Rec, headpol)
s1 := rest s1
headpol := cons([0, 0]$Rec, headpol)

nextPrimitivePoly f ==
n : NNI := degree f
n = 0 => error "polynomial must have positive degree"
-- make f monic
if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
-- if f = fn*X**n + ... + f{i0}*X**(i0) with the fi non-zero
-- then fRepr := [[n,fn], ... , [i0,f{i0}]]
fRepr : Repr := f pretend Repr
fcopy : Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
    fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
    term := [0,0]$Rec
    fcopy := cons(term, fcopy)
fcopy := reverse fcopy
xn : Rec := first fcopy
c0 : GF := term.coeff
l : NNI := lookup(c0)$GF rem sizeGF
n = 1 =>
-- the polynomial X + c is primitive if and only if -c
-- is a primitive element of GF
q1 : NNI := (sizeGF - 1) :: NNI
while l < q1 repeat -- find next c such that -c is primitive
    l := l + 1
c := index(l :: PI)$GF
"failed"

--- Additional text from the document ---

--- Additional text from the document ---
headpol := rest headpol -- [[i2,f{i2}],...,[n,1]]
taillookuplist := cons(j, taillookuplist)
headlookuplist := rest headlookuplist
-- otherwise set f{i1} := index(j)$GF
setelt(first headpol, coeff, index(j :: PI)$GF)
setfirst_!(headlookuplist, j)
if empty? taillookuplist then
  pol := revListToSUP cons(constterm, headpol)
--
-- may be improved by excluding reciprocal polynomials
--
-- primitive? pol => return pol
else
  -- go back to fk
  headpol := cons(first tailpol, headpol) -- [[k,fk],...,[n,1]]
tailpol := rest tailpol
  headlookuplist := cons(first taillookuplist,
  headlookuplist)
taillookuplist := rest taillookuplist
if weight = n then notReady := false
else
  -- must search for polynomial with greater weight
  if empty? s then -- compute s(f)
    restfcopy := rest fcopy
    for entry in restfcopy repeat s := cons(entry.expnt, s)
    s1 := nextSubset(rest s, n1) :: L NNI
    s := cons(0, s1)
    weight := #s
    taillookuplist := []
    headlookuplist := cons(sizeGF, new((weight-2) :: NNI, 1))
tailpol := []
-- headpol = [[s.2,0], [s.3,1], ... , [s.weight,1], [n,1]] :
  headpol := [[first s1, 0]$Rec]
  while not empty? (s1 := rest s1) repeat
    headpol := cons([first s1, 1]$Rec, headpol)
  headpol := reverse cons([n, 1]$Rec, headpol)
-- next polynomial must have greater constant term
  l := l + 1
  c := index(l :: PI)$GF
"failed"

nextNormalPoly f ==
  n : NNI := degree f
  n = 0 => error "polynomial must have positive degree"
  -- make f monic
  if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
  -- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
  -- then fRepr := [[n,fn], ... , [i0,f{i0}]]
  fRepr : Repr := f pretend Repr
  fcopy : Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
  fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
  term := [0,0]$Rec
  fcopy := cons(term, fcopy)
fcopy := reverse fcopy -- [[n,1], [r,fr], ... , [0,f0]]
xn : Rec := first fcopy
middlepol : Repr := rest fcopy -- [[r,fr], ... , [0,f0]]
a0 : GF := (first middlepol).coeff -- fr
l : NNI := lookup(a0)$GF rem sizeGF
n = 1 =>
  -- the polynomial X + a is normal if and only if a is not zero
  l = sizeGF - 1 => "failed"
  [xn, [0, index((l+1) :: PI)$GF]$Rec] pretend SUP
  n1 : NNI := (n - 1) :: NNI
  n2 : NNI := (n1 - 1) :: NNI
-- if the polynomial X**n + a * X**(n-1) + ... is normal then
-- a = -(x + x**q +...+ x**(q**n)) can not be zero (where q = #GF)
a : GF := a0
  -- if a = 0 then set a := 1
  if l = 0 then
    l := 1
    a := 1$GF
  while 1 < sizeGF repeat
    -- (run through the possible values of a)
    if a = a0 then
      -- middlepol = [[0,f0], ... , [m,fm]] with m < n-1
      middlepol := reverse rest middlepol
      weight : NNI := #middlepol -- #s(f) as explained above
      -- the zeroes in the middlelookuplist stand for the fi
      -- whose lookup's were not yet computed :
      middlelookuplist : L NNI := new(weight, 0)
      s : L NNI := [] -- we will compute s(f) only if necessary
      else
        middlepol := [[0,0]$Rec]
        middlelookuplist : L NNI := [sizeGF]
        s : L NNI := [0]
        weight : NNI := 1
        headpol : Repr := [xn, [n1, a]$Rec] -- X**n + a * X**(n-1)
        tailpol : Repr := []
        taillookuplist : L NNI := []
      notReady : Boolean := true
      while notReady repeat
        -- (run through the possible weights)
        while not empty? middlelookuplist repeat
          -- find next polynomial in the above order with fixed
          -- a and weight; assume at this point we have

-- middlepol = [[i1,f{i1}], [i2,f{i2}], ... , [m,fm]] and
-- tailpol = [[k,fk],[...,0,f0]] (with k<i1<i2<...<m)
term := first middlepol
j := first middlelookuplist
if j = 0 then j := lookup(term.coeff)$GF
j := j + 1 -- lookup(f{i1})$GF + 1
j rem sizeGF = 0 =>
-- in this case one has to increase f{i2}
-- tailpol = [[i1,f{i1}],[...,0,f0]]
tailpol := cons(term, tailpol)
middlepol := rest middlepol -- [[i2,f{i2}],[...,m,fm]]
taillookuptlist := cons(j, taillookuptlist)
middlelookuplist := rest middlelookuplist
-- otherwise set f{i1} := index(j)$GF
setelt(first middlepol, coeff, index(j :: PI)$GF)
setfirst_!(middlelookuplist, j)
if empty? taillookuplist then
pol := listToSUP append(headpol, reverse middlepol)
--
-- may be improved by excluding reciprocal polynomials
--
-- normal? pol => return pol
else
-- go back to fk
-- middlepol = [[k,fk],[...,m,fm]]
middlepol := cons(first tailpol, middlepol)
tailpol := rest tailpol
middlelookuptlist := cons(first taillookuptlist, middlelookuplist)
taillookuptlist := rest taillookuptlist
if weight = n1 then notReady := false
else
-- must search for polynomial with greater weight
if empty? s then -- compute s(f)
  restfcopy := rest rest fcopy
  for entry in restfcopy repeat s := cons(entry.expnt, s)
s1 := nextSubset(rest s, n2) :: L NNI
s := cons(0, s1)
weight := #s
taillookuptlist := []
middlelookuplist := cons(sizeGF, new((weight-1) :: NNI, 1))
tailpol := []
-- middlepol = [[0,0], [s.2,1], ... , [s.weight,1]] :
middlepol := []
s1 := reverse s1
while not empty? s1 repeat
  middlepol := cons([[first s1, 1]$Rec, middlepol]
s1 := rest s1
middlepol := cons([0,0]$Rec, middlepol)
-- next polynomial must have greater a
l := l + 1
a := index(l :: PI)$GF
"failed"

nextNormalPrimitivePoly f ==
n : NNI := degree f
n = 0 => error "polynomial must have positive degree"
-- make f monic
if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
-- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
-- then fRepr := [[n,fn], ... , [i0,f{i0}]]
fRepr := Repr := f pretend Repr
fcopy := Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
  fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
  term := [0,0]$Rec
  fcopy := cons(term, fcopy)
fcopy := reverse fcopy -- [[n,1], [r,fr], ... , [0,f0]]
xn : Rec := first fcopy
c0 : GF := term.coeff
lc : NNI := lookup(c0)$GF rem sizeGF
n = 1 =>
  -- the polynomial X + c is primitive if and only if -c
  -- is a primitive element of GF
  q1 : NNI := (sizeGF - 1) :: NNI
  while lc < q1 repeat -- find next c such that -c is primitive
    lc := lc + 1
    c := index(lc :: PI)$GF
    primitive?(-c)$GF =>
      return [xn, [0,c]$Rec] pretend SUP
    "failed"
n1 : NNI := (n - 1) :: NNI
n2 : NNI := (n1 - 1) :: NNI
middlepol : Repr := rest fcopy -- [[r,fr],...,[i0,f{i0}]],[0,f0]]
a0 : GF := (first middlepol).coeff
la : NNI := lookup(a0)$GF rem sizeGF
-- if the polynomial X**n + a * X**(n-1) +...+ c is primitive and
-- normal over GF then (-1)**n * c is a primitive element of GF
-- (cf. [LN] p.90, Th. 3.18), and a = -(x + x**q +...+ x**(q**n))
-- is not zero (where q = #GF)
c : GF := c0
a : GF := a0
-- if a = 0 then set a := 1
if la = 0 then
  la := 1
  a := 1$GF
while lc < sizeGF repeat
  -- (run through the possible values of the constant term)
  noGenerator : Boolean := true
  while noGenerator and lc < sizeGF repeat
    -- find least c >= c0 such that (-1)**n * c0 is primitive
    primitive?((-1)**n * c)$GF => noGenerator := false
    lc := lc + 1
    c := index(lc :: PI)$GF
  noGenerator => return "failed"
  constterm : Rec := [0, c]$Rec
  while la < sizeGF repeat
    -- (run through the possible values of a)
    headpol : Repr := [xn, [n1, a]$Rec] -- X**n + a X**(n-1)
    if c = c0 and a = a0 then
      -- middlepol = [[i0,f{i0}], ..., [m,fm]] with m < n-1
      middlepol := rest reverse rest middlepol
      weight : NNI := #middlepol + 1 -- #s(f)+1 as explained above
      -- the zeroes in the middlelookuplist stand for the fi
      -- whose lookup's were not yet computed :
      middlelookuplist : L NNI := new((weight-1) :: NNI, 0)
      s : L NNI := [] -- we will compute s(f) only if necessary
    else
      pol := listToSUP append(headpol, [constterm])
      normal? pol and primitive? pol => return pol
      middlepol := [[1,0]$Rec]
      middlelookuplist : L NNI := [sizeGF]
      s : L NNI := [0,1]
      weight : NNI := 2
      tailpol : Repr := []
      taillookuplist : L NNI := []
      notReady : Boolean := true
    while notReady repeat
      -- (run through the possible weights)
      while not empty? middlelookuplist repeat
        -- find next polynomial in the above order with fixed
        -- c, a and weight; assume at this point we have
        -- middlepol = [[i1,f{i1}], [i2,f{i2}], ..., [m,fm]]
        -- tailpol = [[k,fk],...,[k0,fk0]] (k0<...<k<i1<...<m)
        term := first middlepol
        j := first middlelookuplist
        if j = 0 then j := lookup(term.coeff)$GF
        j := j + 1 -- lookup(f{i1})$GF + 1
        j rem sizeGF = 0 =>
          -- in this case one has to increase f{i2}
          -- tailpol = [[i1,f{i1}],...,[k0,f{k0}]][k0]<...<k<i1<...<m]
          tailpol := cons(term, tailpol)
          middlepol := rest middlepol -- [[i2,f{i2}],...,[m,fm]]
          middlelookuplist := rest middlelookuplist
        -- otherwise set f{i1} := index(j)$GF
    else
      pol := listToSUP append(headpol, [constterm])
      normal? pol and primitive? pol => return pol
      middlepol := [[1,0]$Rec]
setelt(first middlepol, coeff, index(j :: PI)$GF)
setfirst_!(middlelookuplist, j)
if empty? taillookuplist then
    pol := listToSUP append(headpol, reverse
        cons(constterm, middlepol))
else
    -- may be improved by excluding reciprocal polynomials
    normal? pol and primitive? pol => return pol
    -- go back to fk
    -- middlepol = [[k,fk],...,[m,fn]]
    middlepol := cons(first tailpol, middlepol)
    tailpol := rest tailpol
    middlelookuplist := cons(first taillookuplist,
        middlelookuplist)
    taillookuplist := cons(sizeGF, new((weight-2)::NNI, 1))
    weight := #s
    middlelookuplist := cons(sizeGF, new((weight-2)::NNI, 1))
    tailpol := []
    -- middlepol = [[s.2,0], [s.3,1], ... , [s.weight,1] :
        middlepol := [[first s1, 0]$Rec]
        while not empty? (s1 := rest s1) repeat
            middlepol := cons([first s1, 1]$Rec, middlepol)
        middlepol := reverse middlepol
    -- next polynomial must have greater a
    la := la + 1
    a := index(la :: PI)$GF
    -- next polynomial must have greater constant term
    lc := lc + 1
    c := index(lc :: PI)$GF
    la := 1
    a := 1$GF
    "failed"

nextPrimitiveNormalPoly f == nextNormalPrimitivePoly f
createIrreduciblePoly n ==
    x := monomial(1,1)$SUP
    n = 1 => x
    xn := monomial(1,n)$SUP
n >= sizeGF => nextIrreduciblePoly(xn + x) :: SUP
-- (since in this case there is most no irreducible binomial X+a)
odd? n => nextIrreduciblePoly(xn + 1) :: SUP
nextIrreduciblePoly(xn) :: SUP

createPrimitivePoly n ==
-- (see also the comments in the code of nextPrimitivePoly)
xn := monomial(1,n)$SUP
n = 1 => xn + monomial(-primitiveElement()$GF, 0)$SUP
n := (-1)**n * primitiveElement()$GF
constterm := Rec := [0, c0]$Rec
-- try first (probably faster) the polynomials
-- f = X**n + f{n-1}*X**(n-1) +...+ f1*X + c0 for which
-- fi is 0 or 1 for i=1,...,n-1,
-- and this in the order used to define nextPrimitivePoly
s : L NNI := [0,1]
weight : NNI := 2
s1 : L NNI := [1]
weight := 2
n1 : NNI := (n - 1) :: NNI
notReady : Boolean := true
while notReady repeat
  polRepr : Repr := [constterm]
  while not empty? s1 repeat
    polRepr := cons([first s1, 1]$Rec, polRepr)
    s1 := rest s1
  polRepr := cons([n, 1]$Rec, polRepr)
  -- may be improved by excluding reciprocal polynomials
  -- primitive? (pol := listToSUP polRepr) => return pol
  if weight = n then notReady := false
else
  s1 := nextSubset(rest s, n1) :: L NNI
  s := cons(0, s1)
  weight := #s
-- if there is no primitive f of the above form
-- search now from the beginning, allowing arbitrary
-- coefficients f_i, i = 1,...,n-1
nextPrimitivePoly(xn + monomial(c0, 0)$SUP) :: SUP

createNormalPoly n ==
  n = 1 => monomial(1,1)$SUP + monomial(-1,0)$SUP
  -- get a normal polynomial f = X**n + a * X**(n-1) + ...
  -- with a = -1
  -- [recall that if f is normal over the field GF of order q
  -- then a = -(x + x**q +...+ x**(q**n)) can not be zero;
  -- hence the existence of such an f follows from the
  -- normal basis theorem ([LN] p.60, Th. 2.35) and the
  -- surjectivity of the trace ([LN] p.55, Th. 2.23 (iii))]
  nextNormalPoly(monomial(1,n)$SUP
createNormalPrimitivePoly n ==
xn := monomial(1, n)$SUP
n = 1 => xn + monomial(-primitiveElement()$GF, 0)$SUP
n1 : NNI := (n - 1) :: NNI
c0 : GF := (-1)**n * primitiveElement()$GF
constterm := monomial(c0, 0)$SUP
-- try first the polynomials f = X**n + a * X**(n-1) + ...
-- with a = -1
pol := xn + monomial(-1, n1)$SUP + constterm
normal? pol and primitive? pol => pol
res := nextNormalPrimitivePoly(pol)
res case SUP => res
-- if there is no normal primitive f with a = -1
-- get now one with arbitrary (non-zero) a
-- (the existence is proved in [LS])
pol := xn + monomial(1, n1)$SUP + constterm
normal? pol and primitive? pol => pol
nextNormalPrimitivePoly(pol) :: SUP

createPrimitiveNormalPoly n == createNormalPrimitivePoly n

-- qAdicExpansion m ==
-- ragits : List I := wholeRagits(m :: (RadixExpansion sizeGF))
-- pol : SUP := 0
-- expt : NNI := #ragits
-- for i in ragits repeat
-- expt := (expt - 1) :: NNI
-- if i ^= 0 then pol := pol + monomial(index(i::PI)$GF, expt)
-- pol

-- random == qAdicExpansion(random()$I)

-- random n ==
-- pol := monomial(1, n)$SUP
-- n1 : NNI := (n - 1) :: NNI
-- for i in 0..n1 repeat
-- if (c := random()$GF) ^= 0 then
-- pol := pol + monomial(c, i)$SUP
-- pol

random n ==
polRepr : Repr := []
n1 : NNI := (n - 1) :: NNI
for i in 0..n1 repeat
  if (c := random()$GF) ^= 0 then
    polRepr := cons([i, c]$Rec, polRepr)
c cons([n, 1$GF]$Rec, polRepr) pretend SUP
random(m,n) ==
    if m > n then (m,n) := (n,m)
    d : NNI := (n - m) :: NNI
    if d > 1 then n := ((random()$I rem (d::PI)) + m) :: PI
    random(n)

package FFPOLY2 FiniteFieldPolynomialPackage2

FiniteFieldPolynomialPackage2 examples

FiniteFieldPolynomialPackage2(F,GF) exports some functions concerning
finite fields, which depend on a finite field GF and an algebraic extension F of GF, e.g. a zero of a polynomial over GF in F.

See Also:
- )show FiniteFieldPolynomialPackage2

---

FiniteFieldPolynomialPackage2 (FFPOLY2)

Exports:
rootOfIrreduciblePoly

--- package FFPOLY2 FiniteFieldPolynomialPackage2 ---

)abbrev package FFPOLY2 FiniteFieldPolynomialPackage2
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopaedia of Mathematics and
++ Description:
++ FiniteFieldPolynomialPackage2(F,GF) exports some functions concerning
++ finite fields, which depend on a finite field GF and an
++ algebraic extension F of GF, e.g. a zero of a polynomial
++ over GF in F.

FiniteFieldPolynomialPackage2(F,GF):Exports == Implementation where
  F:FieldOfPrimeCharacteristic with
  coerce: GF -> F
  ++ coerce(x) \undocumented{}
  lookup: F -> PositiveInteger
++ lookup(x) \undocumented{}
basis: PositiveInteger -> Vector F
++ basis(n) \undocumented{}
Frobenius: F -> F
++ Frobenius(x) \undocumented{}
-- F should be a algebraic extension of the finite field GF, either an
-- algebraic closure of GF or a simple algebraic extension field of GF
GF:FiniteFieldCategory

I ==> Integer
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
SUP ==> SparseUnivariatePolynomial
MM ==> ModMonic(GF,SUP GF)
OUT ==> OutputForm
M ==> Matrix
V ==> Vector
L ==> List
FFPOLY ==> FiniteFieldPolynomialPackage(GF)
SUPF2 ==> SparseUnivariatePolynomialFunctions2(GF,F)

Exports ==> with

rootOfIrreduciblePoly:SUP GF -> F
++ rootOfIrreduciblePoly(f) computes one root of the monic,
++ irreducible polynomial f,
++ which degree must divide the extension degree
++ of F over GF,
++ i.e. f splits into linear factors over F.

Implementation ==> add

-- we use berlekamps trace algorithm
-- it is not checked whether the polynomial is irreducible over GF]]
rootOfIrreduciblePoly(pf) ==
-- not irreducible(pf)$FFPOLY =>$
-- error("polynomial has to be irreducible")
sizeGF:=size()$GF
-- if the polynomial is of degree one, we're ready
deg:=degree(pf)$(SUP GF)::PI
deg = 0 => error("no roots")
deg = 1 => -coefficient(pf,0)$(SUP GF)::F
p : SUP F := map(coerce,pf)$SUPF2
-- compute qexp, qexp(i) = x **(size(GF ** i) mod p
-- with this list it's easier to compute the gcd(p(x),trace(x))
qexp:=reducedQPowers(pf)$FFPOLY
stillToFactor:=p
-- take linear independent elements, the basis of F over GF
basis:Vector F:=basis(deg)$F
basispointer:=1
-- as p is irreducible over GF, 0 can't be a root of p
-- therefore we can use the predicate zero?(root) for indicating
-- whether a root is found
root:=0$F
while zero?(root)$F repeat
  beta:=basis.basispointer
  -- gcd(trace(x)+gf,p(x)) has degree 0, that's why we skip beta=1
  if beta = 1$F then
    basispointer:=basispointer + 1
    beta:= basis.basispointer
  basispointer:=basispointer+1
  -- compute the polynomial trace(beta * x) mod p(x) using explist
  trModp:=map(coerce,qexp.0)$SUP F2 * beta
  for i in 1..deg-1 repeat
    beta:=Frobenius(beta)
    trModp:=trModp +$(SUP F) beta *$(SUP F) map(coerce,qexp.i)$SUP F2
  -- if it is of degree 0, it doesn't help us finding a root
  if degree(trModp)$(SUP F) > 0 then
    -- for all elements gf of GF do
    for j in 1..sizeGF repeat
      -- compute gcd(trace(beta * x) + gf, stillToFactor)
      h:=gcd(stillToFactor,trModp +$(SUP F) _
               (index(j pretend PI)$GF::F::(SUP F)))$(SUP F)
      -- make the gcd polynomial monic
      if leadingCoefficient(h)$(SUP F) ^= 1$F then
        h:= (inv leadingCoefficient(h)) * h
        degh:=degree(h)$(SUP F)
        degSTF:=degree(stillToFactor)$(SUP F)
        if degh has degree one we are ready
        degh = 1 =>
        if the quotient of stillToFactor and the gcd has
        -- degree one, we're also ready
        degSTF - degh = 1 =>
        root:= -coefficient(stillToFactor quo h,0)$(SUP F)
        -- otherwise the gcd helps us finding a root, only if its
        -- degree is between 2 and degree(stillToFactor)-2
        if degh > 1 and degh < degSTF then
          2*degh > degSTF =>
          stillToFactor := stillToFactor quo h
          stillToFactor := h
root

— FFPOLY2.dotabb —
"FFPOLY2" -> "IVECTOR"

---

package FFSLPE FiniteFieldSolveLinearPolynomialEquation

--- FiniteFieldSolveLinearPolynomialEquation.input ---

)set break resume
)sys rm -f FiniteFieldSolveLinearPolynomialEquation.output
)spool FiniteFieldSolveLinearPolynomialEquation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldSolveLinearPolynomialEquation
--E 1

)spool
)lisp (bye)

---

--- FiniteFieldSolveLinearPolynomialEquation.help ---

====================================================================
FiniteFieldSolveLinearPolynomialEquation examples
====================================================================

This package solves linear diophantine equations for Bivariate polynomials over finite fields

See Also:
  o )show FiniteFieldSolveLinearPolynomialEquation

---
FiniteFieldSolveLinearPolynomialEquation (FFSLPE)

Exports:
solveLinearPolynomialEquation

— package FFSLPE FiniteFieldSolveLinearPolynomialEquation —

)abbrev package FFSLPE FiniteFieldSolveLinearPolynomialEquation
++ Author: Davenport
++ Date Created: 1991
++ Description:
++ This package solves linear diophantine equations for Bivariate polynomials
++ over finite fields

FiniteFieldSolveLinearPolynomialEquation(F:FiniteFieldCategory,
FP:UnivariatePolynomialCategory F,
FPP:UnivariatePolynomialCategory FP): with

solveLinearPolynomialEquation: (List FPP, FPP) -> Union(List FPP,"failed")
++ solveLinearPolynomialEquation([f1, ..., fn], g)
++ (where the fi are relatively prime to each other)
++ returns a list of ai such that
++ \spad{g/prod fi = sum ai/fi}
++ or returns "failed" if no such list of ai’s exists.

== add
oldlp:List FPP := []
slpePrime: FP := monomial(1,1)
oldtable:Vector List FPP := []
lp: List FPP
p: FPP
import DistinctDegreeFactorize(F,FP)
solveLinearPolynomialEquation(lp,p) ==
if (oldlp ^= lp) then
  -- we have to generate a new table
  deg:= +/[degree u for u in lp]
  ans:Union(Vector List FPP,"failed")="failed"
  slpePrime:=monomial(1,1)+monomial(1,0) -- x+1: our starting guess
  while (ans case "failed") repeat
ans:=tablePow(deg,slpePrime,lp)$GenExEuclid(FP,FPP)
if (ans case "failed") then
    slpePrime:= nextItem(slpePrime)::FP
while (degree slpePrime > 1) and
not irreducible? slpePrime repeat
    slpePrime := nextItem(slpePrime)::FP
oldtable:=(ans:: Vector List FPP)
answer:=solveid(p,slpePrime,oldtable)
anime

-- FFSLPE.dotabb --

"FFSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFSLPE" -> "PFECAT"

package FFSQFR FiniteFieldSquareFreeDecomposition

— FiniteFieldSquareFreeDecomposition.input —

)set break resume
)sys rm -f FiniteFieldSquareFreeDecomposition.output
)spool FiniteFieldSquareFreeDecomposition.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FiniteFieldSquareFreeDecomposition
--R
--R FiniteFieldSquareFreeDecomposition(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory(K)) is a package
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFSQFR
--R
--R----------------------------------- Operations -----------------------------------
--R Musser : PolK -> Factored(PolK)    Yun : PolK -> Factored(PolK)
--R
--E 1
FiniteFieldSquareFreeDecomposition (FFSQFR)

Exports:
PolK Yun

--- package FFSQFR FiniteFieldSquareFreeDecomposition ---

)abbrev package FFSQFR FiniteFieldSquareFreeDecomposition
++ Author: Patrice Naudin and Claude Quitte
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
FiniteFieldSquareFreeDecomposition (K : FiniteFieldCategory, 
    PolK : UnivariatePolynomialCategory (K)) : with 
    Musser : PolK -> Factored (PolK) 
    Yun : PolK -> Factored (PolK) 
    == add 

    p : NonNegativeInteger := characteristic()$K 
    tableOfSquareFreePolynomials := Table (Integer, PolK) 
    oneYunStep2uple := Record ( 
        simpleDecomposition : tableOfSquareFreePolynomials, 
        gcdOfArgumentAndDerivative : PolK ) 

    rawMusser (P : PolK) : Factored(PolK) ==
        Q : PolK := gcd(P, D(P))
        A : PolK := P quo Q 
        decomposition : Factored(PolK) := 1 
        B : PolK
        for i in 1 .. repeat 
            if i rem p ^= 0 then 
                B := gcd(A, Q) 
                decomposition := sqfrFactor(A quo B, i) * decomposition 
                if B = 1 then leave 
                A := B 
                Q := Q quo A 
        if Q ^= 1 then 
            decomposition:=decomposition * rawMusser (charthRoot(Q)::PolK) ** p 
        return decomposition 

    Musser (P : PolK) : Factored(PolK) ==
        degree (P) = 0 => return P::Factored(PolK) 
        if (lcP : K := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P 
        return lcP::PolK * rawMusser (P) 

    oneYunStep (P : PolK) : oneYunStep2uple ==
        C : PolK := D (P) ; A : PolK := gcd(P, C) 
        gcd_P_P' : PolK := A ; B : PolK := P quo A 
        result : tableOfSquareFreePolynomials := empty () 
        i : Integer := 1 
        repeat 
            C := (C quo A) - D(B) 
            if C = 0 then leave 
            A := gcd(B, C) 
            if A ^= 1 then 
                result (i) := A 
                B := B quo A 
            i := i + 1 
        result (i) := B
rawYun (P : PolK) : tableOfSquareFreePolynomials ==
  u : oneYunStep2uple := oneYunStep (P)
  gcd_P_P' : PolK := u.gcdOfArgumentAndDerivative
  U : tableOfSquareFreePolynomials := u.simpleDecomposition
  R : PolK := gcd_P_P'
  for j in indices (U) repeat
    for k in 1 .. j-1 repeat
      R := R quo U(j)
      if R = 1 then return U
  V : tableOfSquareFreePolynomials := rawYun (charthRoot (R)::PolK)
  result : tableOfSquareFreePolynomials := empty ()
  gcd_Uj_Vk : PolK ;
  for k in indices (V) repeat -- boucle 1
    for j in indices (U) | not (U(j) = 1) repeat -- boucle 2
      gcd_Uj_Vk := gcd (U(j), V(k))
      if not (gcd_Uj_Vk = 1) then
        result (j+p*k) := gcd_Uj_Vk
        V (k) := V(k) quo gcd_Uj_Vk
        U (j) := U(j) quo gcd_Uj_Vk
        if V(k) = 1 then leave
      result (p*k):= V (k)
    for j in indices (U) | not (U(j) = 1) repeat -- boucle 3
      result (j) := U (j)
  return result

Yun(P : PolK) : Factored(PolK) ==
  degree (P) = 0 => P::Factored(PolK)
  if (lcP := leadingCoefficient (P)) ^= 1 then P := inv (lcP)*P
  U : tableOfSquareFreePolynomials := rawYun (P)
PFactored : Factored(PolK) := 1
  for i in indices (U) repeat
    PFactored := PFactored * sqfrFactor (U (i), i)
  return (lcP::PolK) * PFactored

— FFSQFR.dotabb —

"FFSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSQFR"]
package FLAGG2 FiniteLinearAggregateFunctions2

--- FiniteLinearAggregateFunctions2.input ---

)set break resume
)sys rm -f FiniteLinearAggregateFunctions2.output
)spool FiniteLinearAggregateFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteLinearAggregateFunctions2
--E 1

)spool
)lisp (bye)

--- FiniteLinearAggregateFunctions2.help ---

====================================================================
FiniteLinearAggregateFunctions2 examples
====================================================================

FiniteLinearAggregateFunctions2 provides functions involving two
FiniteLinearAggregates where the underlying domains might be
different. An example of this might be creating a list of rational
numbers by mapping a function across a list of integers where the
function divides each integer by 1000.

See Also:
o )show FiniteLinearAggregateFunctions2
FiniteLinearAggregateFunctions2 (FLAGG2)

Exports:

map  reduce  scan

— package FLAGG2 FiniteLinearAggregateFunctions2 —

)abbrev package FLAGG2 FiniteLinearAggregateFunctions2
++ Description:
++ FiniteLinearAggregateFunctions2 provides functions involving two
++ FiniteLinearAggregates where the underlying domains might be
++ different. An example of this might be creating a list of rational
++ numbers by mapping a function across a list of integers where the
++ function divides each integer by 1000.

FiniteLinearAggregateFunctions2(S, A, R, B):  
Exports == Implementation where
S, R: Type
A : FiniteLinearAggregate S
B : FiniteLinearAggregate R

Exports ==> with
map : (S -> R, A) -> B
++ map(f,a) applies function f to each member of aggregate
++ \spad{a} resulting in a new aggregate over a
++ possibly different underlying domain.
reduce : ((S, R) -> R, A, R) -> R
++ reduce(f,a,r) applies function f to each
++ successive element of the
++ aggregate \spad{a} and an accumulant initialized to r.
++ For example,
++ \spad{reduce(_,+$Integer,[1,2,3],0)}
++ does \spad{3+(2+(1+0))}. Note that third argument r
++ may be regarded as the
++ identity element for the function f.
scan : ((S, R) -> R, A, R) -> B
++ scan(f,a,r) successively applies
++ \spad{reduce(f,x,r)} to more and more leading sub-aggregates
++ x of aggregate \spad{a}.
++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
++ \spad{scan(f,a,r)} returns
++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.

**Implementation**

```plaintext
if A has ListAggregate(S) then    -- A is a list-oid
    reduce(fn, 1, ident) ==
        empty? 1 => ident
        reduce(fn, rest 1, fn(first 1, ident))

if B has ListAggregate(R) or not(B has shallowlyMutable) then
    -- A is a list-oid, and B is either list-oids or not mutable
    map(f, 1) == construct [f s for s in entries 1]

    scan(fn, 1, ident) ==
        empty? 1 => empty()
        val := fn(first 1, ident)
        concat(val, scan(fn, rest 1, val))

else    -- A is a list-oid, B a mutable array-oid

    map(f, 1) ==
        i := minIndex(w := new(#l,NIL$Lisp)$B)
        for a in entries 1 repeat (qsetelt_!(w, i, f a); i := inc i)
        w

    scan(fn, 1, ident) ==
        i := minIndex(w := new(#l,NIL$Lisp)$B)
        vl := ident
        for a in entries 1 repeat
            vl := qsetelt_!(w, i, fn(a, vl))
            i := inc i
        w

else    -- A is an array-oid

    reduce(fn, v, ident) ==
        val := ident
        for i in minIndex v .. maxIndex v repeat
            val := fn(qelt(v, i), val)
        val

    if B has ListAggregate(R) then    -- A is an array-oid, B a list-oid
        map(f, v) ==
            construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

    scan(fn, v, ident) ==
        w := empty()$B
        for i in minIndex v .. maxIndex v repeat
            ident := fn(qelt(v, i), ident)
            w := concat(ident, w)
```

---
reverse! w

else  -- A and B are array-oid's
if B has shallowlyMutable then -- B is also mutable
map(f, v) ==
    w := new(#v,NIL$Lisp)$B
    for i in minIndex w .. maxIndex w repeat
        qsetelt_!(w, i, f qelt(v, i))
    w

scan(fn, v, ident) ==
    w := new(#v,NIL$Lisp)$B
    vl := ident
    for i in minIndex v .. maxIndex v repeat
        vl := qsetelt_!(w, i, fn(qelt(v, i), vl))
    w

else  -- B non mutable array-oid
map(f, v) ==
    construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

scan(fn, v, ident) ==
    w := empty()$B
    for i in minIndex v .. maxIndex v repeat
        ident := fn(qelt(v, i), ident)
        w := concat(w, ident)
    w

——-

| FLAGG2.dotabb |
| "FLAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLAGG2"] |
| "FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"] |
| "FLAGG2" -> "FLAGG" |

——-

package FLASORT FiniteLinearAggregateSort

— FiniteLinearAggregateSort.input —

)set break resume
)sys rm -f FiniteLinearAggregateSort.output
FiniteLinearAggregateSort (FLASORT)

Exports:
heapSort quickSort shellSort
package FLASORT FiniteLinearAggregateSort

)abbrev package FLASORT FiniteLinearAggregateSort
++ Author: Michael Monagan Sep/88
++ Description:
++ This package exports 3 sorting algorithms which work over
++ FiniteLinearAggregates.
++ Sort package (in-place) for shallowlyMutable Finite Linear Aggregates
-- the following package is only instantiated over %
-- thus shouldn’t be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('FiniteLinearAggregateSort, $mutableDomains)

FiniteLinearAggregateSort(S, V):Exports == Implementation where
    S: Type
    V: FiniteLinearAggregate(S) with shallowlyMutable
    B ==> Boolean
    I ==> Integer

Exports ==> with
    quickSort: ((S, S) -> B, V) -> V
    ++ quickSort(f, agg) sorts the aggregate agg with the ordering function
    ++ f using the quicksort algorithm.
    heapSort : ((S, S) -> B, V) -> V
    ++ heapSort(f, agg) sorts the aggregate agg with the ordering function
    ++ f using the heapsort algorithm.
    shellSort: ((S, S) -> B, V) -> V
    ++ shellSort(f, agg) sorts the aggregate agg with the ordering function
    ++ f using the shellSort algorithm.

Implementation ==> add
    siftUp : ((S, S) -> B, V, I, I) -> Void
    partition: ((S, S) -> B, V, I, I, I) -> I
    QuickSort: ((S, S) -> B, V, I, I) -> V

    quickSort(l, r) == QuickSort(l, r, minIndex r, maxIndex r)
    siftUp(l, r, i, n) ==
        t := qelt(r, i)
        while (j := 2*i+1) < n repeat
            if (k := j+1) < n and l(qelt(r, j), qelt(r, k)) then j := k
            if l(t, qelt(r,j)) then
                qsetelt_!(r, i, qelt(r, j))
                qsetelt_!(r, j, t)
                i := j
            else leave
heapSort(l, r) ==
  not zero? minIndex r => error "not implemented"
  n := (#r)::I
  for k in shift(n,-1) - 1 .. 0 by -1 repeat siftUp(l, r, k, n)
  for k in n-1 .. 1 by -1 repeat
    swap!(r, 0, k)
    siftUp(l, r, 0, k)
  r

partition(l, r, i, j, k) ==
  -- partition r[i..j] such that r.s <= r.k <= r.t
  x := qelt(r, k)
  t := qelt(r, i)
  qsetelt_!(r, k, qelt(r, j))
  while i < j repeat
    if l(x, t) then
      qsetelt_!(r, j, t)
      j := j-1
      t := qsetelt_!(r, i, qelt(r, j))
    else (i := i+1; t := qelt(r, i))
    qsetelt_!(r, j, x)
  j

QuickSort(l, r, i, j) ==
  n := j - i
  if one? n and l(qelt(r, j), qelt(r, i)) then swap!(r, i, j)
  if (n = 1) and l(qelt(r, j), qelt(r, i)) then swap!(r, i, j)
  n < 2 => return r
  -- for the moment split at the middle item
  k := partition(l, r, i, j, i + shift(n,-1))
  QuickSort(l, r, i, k - 1)
  QuickSort(l, r, k + 1, j)

shellSort(l, r) ==
  m := minIndex r
  n := maxIndex r
  -- use Knuths gap sequence: 1,4,13,40,121,...
  g := 1
  while g <= (n-m) repeat g := 3*g+1
  g := g quo 3
  while g > 0 repeat
    for i in m+g..n repeat
      j := i-g
      while j >= m and l(qelt(r, j+g), qelt(r, j)) repeat
        swap!(r(j,j+g))
        j := j-g
      g := g quo 3
  r
package FSAGG2 FiniteSetAggregateFunctions2

--- FiniteSetAggregateFunctions2.input ---

)set break resume
)sys rm -f FiniteSetAggregateFunctions2.output
)spool FiniteSetAggregateFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteSetAggregateFunctions2
--E 1

)spool
)lisp (bye)

--- FiniteSetAggregateFunctions2.help ---

====================================================================
FiniteSetAggregateFunctions2 examples
====================================================================

FiniteSetAggregateFunctions2 provides functions involving two
finite set aggregates where the underlying domains might be
different. An example of this is to create a set of rational
numbers by mapping a function across a set of integers, where the
function divides each integer by 1000.

See Also:
o )show FiniteSetAggregateFunctions2
FiniteSetAggregateFunctions2 (FSAGG2)

Exports:
map  reduce  scan

— package FSAGG2 FiniteSetAggregateFunctions2 —

)abbrev package FSAGG2 FiniteSetAggregateFunctions2
++ Author: Robert S. Sutor
++ Date Created: 15 May 1990
++ Date Last Updated: 14 Oct 1993
++ Description:
++ FiniteSetAggregateFunctions2 provides functions involving two
++ finite set aggregates where the underlying domains might be
++ different. An example of this is to create a set of rational
++ numbers by mapping a function across a set of integers, where the
++ function divides each integer by 1000.

FiniteSetAggregateFunctions2(S, A, R, B): Exports == Implementation where
S, R: SetCategory
A : FiniteSetAggregate S
B : FiniteSetAggregate R

Exports ==> with
map : (S -> R, A) -> B
++ map(f,a) applies function f to each member of
++ aggregate \spad{a}, creating a new aggregate with
++ a possibly different underlying domain.
reduce : ((S, R) -> R, A, R) -> R
++ reduce(f,a,r) applies function f to each
++ successive element of the aggregate \spad{a} and an
++ accumulant initialised to r.
++ For example, 
++ \spad{reduce(_+$Integer,[2,3],0)} 
++ does a \spad{3+(2+(1+0))}. 
++ Note that third argument r may be regarded 
++ as an identity element for the function.

\texttt{scan : ((S, R) \to R, A, R) \to B} 
++ \spad{scan(f,a,r)} successively applies \spad{reduce(f,x,r)} 
++ to more and more leading sub-aggregates x of 
++ aggregate \spad{a}. 
++ More precisely, if \spad{a} is \spad{[[a1,a2,...]]}, then 
++ \spad{scan(f,a,r)} returns 
++ \spad{[[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]]}.

\texttt{Implementation ==> add}

\texttt{map(fn, a) ==} 
\texttt{set(map(fn, parts a)$ListFunctions2(S, R))$B}
\texttt{reduce(fn, a, ident) ==} 
\texttt{reduce(fn, parts a, ident)$ListFunctions2(S, R)}
\texttt{scan(fn, a, ident) ==} 
\texttt{set(scan(fn, parts a, ident)$ListFunctions2(S, R))$B}

--------------------

--- FSAGG2.dotabb ---

"FSAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSAGG2"] 
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"FSAGG2" -> "FSAGG"

--------------------

package FLOATCP FloatingComplexPackage

--- FloatingComplexPackage.input ---

\texttt{)}set break resume
\texttt{)}sys rm -f FloatingComplexPackage.output
\texttt{)}spool FloatingComplexPackage.output
\texttt{)}set message test on
\texttt{)}set message auto off
\texttt{)}clear all

--S 1 of 1
\texttt{)}show FloatingComplexPackage

--E 1
FloatingComplexPackage examples

This is a package for the approximation of complex solutions for systems of equations of rational functions with complex rational coefficients. The results are expressed as either complex rational numbers or complex floats depending on the type of the precision parameter which can be either a rational number or a floating point number.

See Also:
o )show FloatingComplexPackage

Exports:
complexRoots  complexSolve

---
+++ Date Created: January 1990
+++ Description:
+++ This is a package for the approximation of complex solutions for
+++ systems of equations of rational functions with complex rational
+++ coefficients. The results are expressed as either complex rational
+++ numbers or complex floats depending on the type of the precision
+++ parameter which can be either a rational number or a floating point number.

FloatingComplexPackage(Par): Cat == Cap where
Par : Join(Field, OrderedRing)
K ==> GI
FPK ==> Fraction P K
C ==> Complex
I ==> Integer
NNI ==> NonNegativeInteger
P ==> Polynomial
EQ ==> Equation
L ==> List
SUP ==> SparseUnivariatePolynomial
RN ==> Fraction Integer
NF ==> Float
CF ==> Complex Float
GI ==> Complex Integer
GRN ==> Complex RN
SE ==> Symbol
RFI ==> Fraction P I
INFSP ==> InnerNumericFloatSolvePackage

Cat == with

complexSolve: (L FPK,Par) -> L L EQ P C Par
++ complexSolve(lp,eps) finds all the complex solutions to
++ precision eps of the system lp of rational functions
++ over the complex rationals with respect to all the
++ variables appearing in lp.

complexSolve: (L EQ FPK,Par) -> L L EQ P C Par
++ complexSolve(leq,eps) finds all the complex solutions
++ to precision eps of the system leq of equations
++ of rational functions over complex rationals
++ with respect to all the variables appearing in lp.

complexSolve: (FPK,Par) -> L EQ P C Par
++ complexSolve(p,eps) find all the complex solutions of the
++ rational function p with complex rational coefficients
++ with respect to all the variables appearing in p,
++ with precision eps.

complexSolve: (EQ FPK,Par) -> L EQ P C Par
++ complexSolve(eq,eps) finds all the complex solutions of the
++ equation eq of rational functions with rational rational coefficients
++ with respect to all the variables appearing in eq,
++ with precision eps.

complexRoots : (FPK,Par) -> L C Par
++ complexRoots(rf, eps) finds all the complex solutions of a
++ univariate rational function with rational number coefficients.
++ The solutions are computed to precision eps.

complexRoots : (L FPK,L SE,Par) -> L L C Par
++ complexRoots(lrf, lv, eps) finds all the complex solutions of a
++ list of rational functions with rational number coefficients
++ with respect the the variables appearing in lv.
++ Each solution is computed to precision eps and returned as
++ list corresponding to the order of variables in lv.

Cap == add

-- find the complex zeros of an univariate polynomial --
complexRoots(q:FPK,eps:Par) : L C Par ==
p:=numer q
complexZeros(univariate p,eps)$ComplexRootPackage(SUP GI, Par)

-- find the complex zeros of an univariate polynomial --
complexRoots(lp:L FPK,lv:L SE,eps:Par) : L L C Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p) ^=1]
innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)

complexSolve(lp:L FPK,eps : Par) : L L EQ P C Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p) ^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden[]=[] then
  lv:="setUnion"/[variables dp for dp in lden])
[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
  for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

complexSolve(le:L EQ FPK,eps : Par) : L L EQ P C Par ==
lp:=[lhs ep - rhs ep for ep in le]
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p) ^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden[]=[] then
  lv:="setUnion"/[variables dp for dp in lden])
[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
  for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

complexSolve(p : FPK,eps : Par) : L EQ P C Par ==
(mvar := mainVariable numer p ) case "failed" =>
   error "no variable found"

x:P C Par:=mvar::SE::(P C Par)
[equation(x,val::(P C Par)) for val in complexRoots(p,eps)]

complexSolve(eq : EQ FPK,eps : Par) : L EQ P C Par ==
complexSolve(lhs eq - rhs eq,eps)

---

— FLOATCP.dotabb —

"FLOATCP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLOATCP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"FLOATCP" -> "COMPCAT"

---

package FLOATRP FloatingRealPackage

--- FloatingRealPackage.input ---

)set break resume
)sys rm -f FloatingRealPackage.output
)spool FloatingRealPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FloatingRealPackage
--E 1

)spool
)lisp (bye)

---

— FloatingRealPackage.help —

====================================================================
FloatingRealPackage examples
====================================================================
This is a package for the approximation of real solutions for systems of polynomial equations over the rational numbers. The results are expressed as either rational numbers or floats depending on the type of the precision parameter which can be either a rational number or a floating point number.

See Also:
0 )show FloatingRealPackage

---

FloatingRealPackage (FLOATRP)

Exports:
realRoots solve

--- package FLOATRP FloatingRealPackage ---

)abbrev package FLOATRP FloatingRealPackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Description:
++ This is a package for the approximation of real solutions for
++ systems of polynomial equations over the rational numbers.
++ The results are expressed as either rational numbers or floats
++ depending on the type of the precision parameter which can be
++ either a rational number or a floating point number.

FloatingRealPackage(Par): Cat == Cap where
I    ==> Integer
NNI  ==> NonNegativeInteger
P    ==> Polynomial
EQ ==> Equation
L ==> List
SUP ==> SparseUnivariatePolynomial
RN ==> Fraction Integer
NF ==> Float
CF ==> Complex Float
GI ==> Complex Integer
GRN ==> Complex RN
SE ==> Symbol
RFI ==> Fraction P I
INFSP ==> InnerNumericFloatSolvePackage

Par : Join(OrderedRing, Field) -- RN or NewFloat

Cat == with

solve: (L RFI,Par) -> L L EQ P Par
++ solve(lp,eps) finds all of the real solutions of the
++ system lp of rational functions over the rational numbers
++ with respect to all the variables appearing in lp,
++ with precision eps.

solve: (L EQ RFI,Par) -> L L EQ P Par
++ solve(leq,eps) finds all of the real solutions of the
++ system leq of equations of rational functions
++ with respect to all the variables appearing in lp,
++ with precision eps.

solve: (RFI,Par) -> L EQ P Par
++ solve(p,eps) finds all of the real solutions of the
++ univariate rational function p with rational coefficients
++ with respect to the unique variable appearing in p,
++ with precision eps.

solve: (EQ RFI,Par) -> L EQ P Par
++ solve(eq,eps) finds all of the real solutions of the
++ univariate equation eq of rational functions
++ with respect to the unique variables appearing in eq,
++ with precision eps.

realRoots: (L RFI,L SE,Par) -> L L Par
++ realRoots(lp,lv,eps) computes the list of the real
++ solutions of the list lp of rational functions with rational
++ coefficients with respect to the variables in lv,
++ with precision eps. Each solution is expressed as a list
++ of numbers in order corresponding to the variables in lv.

realRoots : (RFI,Par) -> L Par
++ realRoots(rf, eps) finds the real zeros of a univariate
++ rational function with precision given by eps.
Cap == add

makeEq(nres:L Par,lv:L SE) : L EQ P Par ==
[equation(x::(P Par),r::(P Par)) for x in lv for r in nres]

-- find the real zeros of an univariate rational polynomial --
realRoots(p:RFI,eps:Par) : L Par ==
innerSolve(numer p,eps)$INFSP(I,Par,Par)

-- real zeros of the system of polynomial lp --
realRoots(lp:L RFI,lv:L SE,eps: Par) : L L Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p)^=1]
innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)

solve(lp:L RFI,eps : Par) : L L EQ P Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
lv:="setUnion(lv,"setUnion"/[variables dp for dp in lden])
[makeEq(numeres,lv) for numeres
 in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(le:L EQ RFI,eps : Par) : L L EQ P Par ==
lp:=[lhs ep - rhs ep for ep in le]
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
lv:="setUnion(lv,"setUnion"/[variables dp for dp in lden])
[makeEq(numeres,lv) for numeres
 in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(p : RFI,eps : Par) : L EQ P Par ==
(mvar := mainVariable numer p ) case "failed" =>
error "no variable found"
x:P Par:=mvar::SE::(P Par)
[equation(x,val::(P Par)) for val in realRoots(p,eps)]

solve(eq : EQ RFI,eps : Par) : L EQ P Par ==
solve(lhs eq - rhs eq,eps)
package FCPAK1 FortranCodePackage1

— FortranCodePackage1.input —

)set break resume
)sys rm -f FortranCodePackage1.output
)spool FortranCodePackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranCodePackage1
--E 1

)spool
)lisp (bye)

— FortranCodePackage1.help —

====================================================================
FortranCodePackage1 examples
====================================================================

FortranCodePackage1 provides some utilities for producing useful objects in FortranCode domain.

The Package may be used with the FortranCode domain and its printCode or possibly via an outputAsFortran.

The package provides items of use in connection with ASPs in the AXIOM-NAG link and, where appropriate, naming accords with that in IRENA.

The easy-to-use functions use Fortran loop variables I1, I2, and it is users' responsibility to check that this is sensible. The advanced functions use SegmentBinding to allow users control...
See Also:
o )show FortranCodePackage1

---

FortranCodePackage1 (FCPAK1)

Exports:
identitySquareMatrix zeroMatrix zeroSquareMatrix zeroVector

---

)abbrev package FCPAK1 FortranCodePackage1
++ Author: Grant Keady and Godfrey Nolan
++ Date Created: April 1993
++ Description:
++ \spadtype{FortranCodePackage1} provides some utilities for
++ producing useful objects in FortranCode domain.
++ The Package may be used with the FortranCode domain and its
++ \spad{printCode} or possibly via an outputAsFortran.
++ (The package provides items of use in connection with ASPs
++ in the AXIOM-NAG link and, where appropriate, naming accords
++ with that in IRENA.)
++ The easy-to-use functions use Fortran loop variables I1, I2,
++ and it is users’ responsibility to check that this is sensible.
++ The advanced functions use SegmentBinding to allow users control
++ over Fortran loop variable names.
-- Later might add functions to build
-- diagonalMatrix from List, i.e. the FC version of the corresponding
-- AXIOM function from MatrixCategory;
-- bandedMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedMatrix Domain
-- bandedSymmetricMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedSymmetricMatrix Domain

FortranCodePackage1: Exports == Implementation where

NNI ==> NonNegativeInteger
PI ==> PositiveInteger
PIN ==> Polynomial(Integer)
SBINT ==> SegmentBinding(Integer)
SEGIN ==> Segment(Integer)
LSBINT ==> List(SegmentBinding(Integer))
SBPIN ==> SegmentBinding(Polynomial(Integer))
SEGPIN ==> Segment(Polynomial(Integer))
LSBPIN ==> List(SegmentBinding(Polynomial(Integer)))
FC ==> FortranCode
EXPRESSION ==> Union(Expression Integer,Expression Float,Expression Complex Integer,Expression Complex Float)

Exports == with

zeroVector: (Symbol,PIN) -> FC
++ zeroVector(s,p) undocumented{}

zeroMatrix: (Symbol,PIN,PIN) -> FC
++ zeroMatrix(s,p,q) uses loop variables in the Fortran, I1 and I2

zeroMatrix: (Symbol,SBPIN,SBPIN) -> FC
++ zeroMatrix(s,b,d) in this version gives the user control
++ over names of Fortran variables used in loops.

zeroSquareMatrix: (Symbol,PIN) -> FC
++ zeroSquareMatrix(s,p) undocumented{}

identitySquareMatrix: (Symbol,PIN) -> FC
++ identitySquareMatrix(s,p) undocumented{}

Implementation ==> add
import FC

zeroVector(fname:Symbol,n:PIN):FC ==
ue:Expression(Integer) := 0
i1:Symbol := "$I1":Symbol
lp1:PIN := 1::PIN
hp1:PIN := n
segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
segbp1:SBPIN := equation(i1,segp1)$SBPIN
ipl:PIN := i1::PIN
indices:List(PIN) := [ipl]
fa:FC := forLoop(segbp1,assign(fname,indices,ue)$FC)$FC
fa
zeroMatrix(fname:Symbol,m:PIN,n:PIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := m
  segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
  sgbp1:SBPIN := equation(i1,segp1)$SBPIN
  i2:Symbol := "I2"::Symbol
  hp2:PIN := n
  segp2:SEGPIN := segment(lp1,hp2)$SEGPIN
  sgbp2:SBPIN := equation(i2,segp2)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

zeroMatrix(fname:Symbol,segbp1:SBPIN,segbp2:SBPIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := variable(segbp1)$SBPIN
  i2:Symbol := variable(segbp2)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

zeroSquareMatrix(fname:Symbol,n:PIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := n
  segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
  sgbp1:SBPIN := equation(i1,segp1)$SBPIN
  i2:Symbol := "I2"::Symbol
  segp2:SEGPIN := equation(i2,segp1)$SEGPIN
  sgbp2:SBPIN := equation(i2,segp2)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

identitySquareMatrix(fname:Symbol,n:PIN):FC ==
  ue:Expression(Integer) := 0
  u1:Expression(Integer) := 1
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := n
  segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
package FOP FortranOutputStackPackage

— FortranOutputStackPackage.input —

)set break resume
)sys rm -f FortranOutputStackPackage.output
)spool FortranOutputStackPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranOutputStackPackage
--E 1

)spool
)lisp (bye)

— FortranOutputStackPackage.help —
FortranOutputStackPackage examples
====================================================================
Code to manipulate Fortran Output Stack

See Also:
- )show FortranOutputStackPackage

---

FortranOutputStackPackage (FOP)

Exports:
- popFortranOutputStack
- clearFortranOutputStack
- pushFortranOutputStack
- pushFortranOutputStack
- showFortranOutputStack
- topFortranOutputStack

---

)abbrev package FOP FortranOutputStackPackage
-- Because of a bug in the compiler:
)bo $noSubsumption:=false

++ Author: Mike Dewar
++ Date Created: October 1992
++ Description:
++ Code to manipulate Fortran Output Stack

FortranOutputStackPackage() : specification == implementation where

specification == with

  clearFortranOutputStack : () -> Stack String
    ++ clearFortranOutputStack() clears the Fortran output stack
showFortranOutputStack : () -> Stack String
++ showFortranOutputStack() returns the Fortran output stack

popFortranOutputStack : () -> Void
++ popFortranOutputStack() pops the Fortran output stack

pushFortranOutputStack : FileName -> Void
++ pushFortranOutputStack(f) pushes f onto the Fortran output stack

pushFortranOutputStack : String -> Void
++ pushFortranOutputStack(f) pushes f onto the Fortran output stack

topFortranOutputStack : () -> String
++ topFortranOutputStack() returns the top element of the Fortran output stack

implementation == add

import MoreSystemCommands

-- A stack of filenames for Fortran output. We are sharing this with
-- the standard Fortran output code, so want to be a bit careful about
-- how we interact with what the user does independently. We get round
-- potential problems by always examining the top element of the stack
-- before we push. If the user has redirected output then we alter our
-- top value accordingly.

fortranOutputStack : Stack String := empty()@(Stack String)

topFortranOutputStack():String == string(_$fortranOutputFile$Lisp)

pushFortranOutputStack(fn:FileName):Void ==
  if empty? fortranOutputStack then
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
  else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
    pop! fortranOutputStack
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    push!( fn::String,fortranOutputStack)
    systemCommand concat(["set output fortran quiet ", fn::String])$String void()

pushFortranOutputStack(fn:String):Void ==
  if empty? fortranOutputStack then
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
  else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
    pop! fortranOutputStack
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    push!( fn,fortranOutputStack)
    systemCommand concat(["set output fortran quiet ", fn])$String void()

popFortranOutputStack():Void ==
  if not empty? fortranOutputStack then pop! fortranOutputStack
  if empty? fortranOutputStack then push!("CONSOLE",fortranOutputStack)
  systemCommand concat(["set output fortran quiet append ",_
package FORT FortranPackage

---

FortranPackage.input ---

)set break resume
)sys rm -f FortranPackage.output
)spool FortranPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranPackage
--E 1

)spool
)lisp (bye)

---

FortranPackage.help ---

====================================================================
FortranPackage examples
This package provides an interface to the boot code for calling Fortran

See Also:
- )show FortranPackage

---

**FortranPackage (FORT)**

Exports:
- outputAsFortran
- linkToFortran
- setLegalFortranSourceExtensions

--- package FORT FortranPackage ---

)abbrev package FORT FortranPackage
-- Because of a bug in the compiler:
)bo $noSubsumption:=true

++ Author: Mike Dewar
++ Date Created: October 6 1991
++ Date Last Updated: 13 July 1994
++ Description:
++ provides an interface to the boot code for calling Fortran

FortranPackage(): Exports == Implementation where
FST ==> FortranScalarType
SEX ==> SExpression
L ==> List
S ==> Symbol
FOP ==> FortranOutputStackPackage
U ==> Union(array:L S,scalar:S)
Exports ==> with
  linkToFortran: (S, L U, L L U, L S) -> SEX
    ++ linkToFortran(s,1,l,l,v) \ undocumented{}
  linkToFortran: (S, L U, L L U, L S, S) -> SEX
    ++ linkToFortran(s,1,l,l,v,t) \ undocumented{}
  linkToFortran: (S, L S, TheSymbolTable, L S) -> SEX
    ++ linkToFortran(s,l,t,l,v) \ undocumented{}
outputAsFortran: FileName -> Void
  ++ outputAsFortran(fn) \ undocumented{}
setLegalFortranSourceExtensions: List String -> List String
  ++ setLegalFortranSourceExtensions(l) \ undocumented{}

Implementation ==> add

legalFortranSourceExtensions : List String := ["f"]

setLegalFortranSourceExtensions(l:List String):List String ==
  legalFortranSourceExtensions := l

checkExtension(fn : FileName): String ==
  -- Does it end in a legal extension ?
  stringFn := fn::String
  not member?(extension fn,legalFortranSourceExtensions) =>
    error [stringFn,"is not a legal Fortran Source File."]
  stringFn

outputAsFortran(fn:FileName):Void ==
  -- source : String := checkExtension fn
  source : String := fn::String
  if not readable? fn =>
    popFortranOutputStack()$FOP
    error([source,"is not readable"]@List(String))
  target : String := topFortranOutputStack()$FOP
  command : String :=
    concat(['"sys rm -f ",target," ; cp ",source," ",target"])$String
  systemCommand(command)$MoreSystemCommands
  void()$Void

linkToFortran(name:S, args:L U, decls:L L U, res:L(S)):SEX ==
  makeFort(name, args, decls, res, NIL$Lisp, NIL$Lisp)$Lisp
linkToFortran(name:S, args:L U, decls:L L U, res:L(S), returnType:S):SEX ==
  makeFort(name, args, decls, res, returnType, NIL$Lisp)$Lisp
dimensions(type:FortranType):SEX ==
  convert([convert(convert(u)@InputForm)@SEX |
    for u in dimensionsOf(type)])$SEX
ftype(name:S, type:FortranType):SEX ==
makeAspList(asp:S,syms:TheSymbolTable):SExpression==
symtab : SymbolTable := symbolTableOf(asp,syms)
[asp,returnTypeOf(asp,syms),argumentListOf(asp,syms), _
 [ftype(u,fortranTypeOf(u,symtab)) for u in parametersOf symtab]]$Lisp

linkToFortran(name:S,aArgs:L S,syms:TheSymbolTable,res:L S):SEX ==
arguments : L S := argumentListOf(name,syms)$TheSymbolTable
dummies : L S := setDifference(arguments,aArgs)
symbolTable:SymbolTable := symbolTableOf(name,syms)
symbolList := newTypeLists(symbolTable)
rt:Union(fst: FST,void: "void") := returnTypeOf(name,syms)$TheSymbolTable

-- Look for arguments which are subprograms
asps :=[makeAspList(u,syms) for u in externalList(symbolTable)$SymbolTable]
rt case fst =>
makeFort1(name,arguments,aArgs,dummies,symbolList,res,(rt.fst)::S,asps)$Lisp
makeFort1(name,arguments,aArgs,dummies,symbolList,res,NIL$Lisp,asps)$Lisp

package FRIDEAL2 FractionalIdealFunctions2

— FractionalIdealFunctions2.input —

)set break resume
)sys rm -f FractionalIdealFunctions2.output
)spool FractionalIdealFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionalIdealFunctions2
--E 1
FractionalIdealFunctions2 (FRIDEAL2)

Exports:
map

— package FRIDEAL2 FractionalIdealFunctions2 —

)abbrev package FRIDEAL2 FractionalIdealFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1 Feb 1989
++ Date Last Updated: 27 Feb 1990
++ Description:
++ Lifting of morphisms to fractional ideals.
FractionalIdealFunctions2(R1, F1, U1, A1, R2, F2, U2, A2):
Exports == Implementation where
  R1, R2: EuclideanDomain
  F1: QuotientFieldCategory R1
  U1: UnivariatePolynomialCategory F1
  A1: Join(FramedAlgebra(F1, U1), RetractableTo F1)
  F2: QuotientFieldCategory R2
  U2: UnivariatePolynomialCategory F2
  A2: Join(FramedAlgebra(F2, U2), RetractableTo F2)

Exports ==> with
  map: (R1 -> R2, FractionalIdeal(R1, F1, U1, A1)) -> FractionalIdeal(R2, F2, U2, A2)
    ++ map(f, i) \undocumented{}

Implementation ==> add
  fmap: (F1 -> F2, A1) -> A2
  fmap(f, a) ==
    v := coordinates a
    represents
    [f qelt(v, i) for i in minIndex v .. maxIndex v]$Vector(F2)
  map(f, i) ==
    b := basis i
    ideal [fmap(s +-> f(numer s) / f(denom s), qelt(b, j))
      for j in minIndex b .. maxIndex b]$Vector(A2)

— FRIDEAL2.dotabb —
"FRIDEAL2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRIDEAL2"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FRIDEAL2" -> "FRAMALG"

package FFFG FractionFreeFastGaussian

— FractionFreeFastGaussian.input —

)set break resume
This package implements the interpolation algorithm proposed in Beckermann, Bernhard and Labahn, George, Fraction-free computation of matrix rational interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and Applications 22.

The packages defined in this file provide fast fraction free rational interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

See Also:
  o )show FractionFreeFastGaussian

— FractionFreeFastGaussian.help —

====================================================================
FractionFreeFastGaussian examples
====================================================================
FractionFreeFastGaussian (FFFG)

Exports:
  \text{DiffAction, DiffC, ShiftAction, ShiftC, fffg, generalCoefficient, generalInterpolation, interpolate, qShiftAction, qShiftC}

--- package FFFG FractionFreeFastGaussian ---

)abbrev package FFFG FractionFreeFastGaussian
++ Author: Martin Rubey
++ Description:
++ This package implements the interpolation algorithm proposed in Beckermann,
++ Bernhard and Labahn, George, Fraction-free computation of matrix rational
++ interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and
++ Applications 22.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussian(D, V): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: AbelianMonoidRing(D, NonNegativeInteger) -- for example, SUP D

SUP => SparseUnivariatePolynomial

cFunction => (NonNegativeInteger, Vector SUP D) -> D

CoeffAction => (NonNegativeInteger, NonNegativeInteger, V) -> D

Exports == with

  fffg: (List D, cFunction, List NonNegativeInteger) -> Matrix SUP D
++ \text{\textbackslash spad} fffg \text{\textbackslash is the general algorithm as proposed by Beckermann and}
++ Labahn.
++
++ The first argument is the list of \text{c\_\{i,i\}}. These are the only values
The following function could be moved to FFFGF, parallel to \texttt{generalInterpolation}. However, the reason for moving \texttt{generalInterpolation} for fractions to a separate package was the need of a generic signature, hence the extra argument \texttt{VF} to FFFGF. In the special case of rational interpolation, this extra argument is not necessary, since we are always returning a fraction of SUPs, and ignore \texttt{V}. In fact, \texttt{V} is not needed for \texttt{fffg} itself, only if we want to specify a \texttt{CoeffAction}.

Thus, maybe it would be better to move \texttt{fffg} to a separate package?

--- package FFFG FractionFreeFastGaussian ---

\texttt{interpolate}: (List $D$, List $D$, NonNegativeInteger) $\rightarrow$ Fraction SUP $D$
\texttt{generalInterpolation}: (List $D$, CoeffAction, Vector $V$, List NonNegativeInteger) $\rightarrow$ Matrix SUP $D$
++ eta + e.i - [1,1,...,1], where the degree of zero is -1.
++
++ The first argument C is the list of coefficients c_{k,k} in the
++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
++
++ The second argument, CA(k, 1, f), should return the coefficient of x^k
++ in z^l f(x).

\text{generalInterpolation}: (\text{List D, CoeffAction},
\text{Vector V, NonNegativeInteger, NonNegativeInteger})
\rightarrow \text{Stream Matrix SUP D}
++ \text{generalInterpolation}(C, CA, f, sumEta, maxEta) applies
++ \text{generalInterpolation}(C, CA, f, eta) for all possible \text{eta}
++ with maximal entry \text{maxEta} and sum of entries at most
++ \text{sumEta}.
++
++ The first argument C is the list of coefficients c_{k,k} in the
++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
++
++ The second argument, CA(k, 1, f), should return the coefficient of x^k
++ in z^l f(x).

\text{generalCoefficient}: (\text{CoeffAction, Vector V,}
\text{NonNegativeInteger, Vector SUP D}) \rightarrow \text{D}
++ \text{generalCoefficient}(action, f, k, p) gives the coefficient of
++ x^k in p(z) \dot f(x), where the action of z^l on a polynomial in x is
++ given by action, i.e., action(k, l, f) should return the coefficient
++ of x^k in z^l f(x).

\text{ShiftAction}: (\text{NonNegativeInteger, NonNegativeInteger, V}) \rightarrow \text{D}
++ \text{ShiftAction}(k, l, g) gives the coefficient of x^k in z^l g(x),
++ where \text{z*(a+b*x+c*x^2+d*x^3+...)} = (b*x+2*c*x^2+3*d*x^3+...). In
++ terms of sequences, z*u(n)=n*u(n).

\text{ShiftC}: \text{NonNegativeInteger} \rightarrow \text{List D}
++ \text{ShiftC} gives the coefficients c_{k,k} in the expansion <x^k> z
++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
++ shifting. In fact, the result is [0,1,2,...]

\text{DiffAction}: (\text{NonNegativeInteger, NonNegativeInteger, V}) \rightarrow \text{D}
++ \text{DiffAction}(k, l, g) gives the coefficient of x^k in z^l g(x),
++ where z*(a+b*x+c*x^2+d*x^3+...) = (a*x+b*x^2+c*x^3+...), i.e.,
++ multiplication with x.

\text{DiffC}: \text{NonNegativeInteger} \rightarrow \text{List D}
++ \text{DiffC} gives the coefficients c_{k,k} in the expansion <x^k> z
++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
++ shifting. In fact, the result is [0,0,0,...]

\text{qShiftAction}: (\text{D, NonNegativeInteger, NonNegativeInteger, V}) \rightarrow \text{D}
++ \spad{qShiftAction(q, k, l, g)} gives the coefficient of \(x^k\) in \(z^l\) 
++ \(g(x)\), where 
++ \(z*(a+b*x+c*x^2+d*x^3+...)=\) 
++ \((a+q*b*x+q^2*c*x^2+q^3*d*x^3+...).\) In terms of sequences, 
++ \(z*u(n)=q^n*u(n).\)

\[
qShiftC: (D, \text{NonNegativeInteger}) \to \text{List D}
\]
++ \spad{qShiftC} gives the coefficients \(c_{k,k}\) in the expansion \(\langle x^k\rangle z\)
++ \(g(x) = \sum_{i=0}^k c_{k,i} \langle x^i\rangle g(x)\), where \(z\) acts on \(g(x)\) by
++ shifting. In fact, the result is \([1,q,q^2,...]\)

Implementation ==> add

-- Shift Operator

-- ShiftAction(k, l, f) is the CoeffAction appropriate for the shift operator.

\[
\text{ShiftAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==}
\]
\[
k**l*\text{coefficient}(f, k)
\]

\[
\text{ShiftC(total: NonNegativeInteger): List D ==}
\]
\[
[\cdot D \text{ for i in } 0..\text{total}-1]
\]

-- q-Shift Operator

-- q-ShiftAction(k, l, f) is the CoeffAction appropriate for the q-shift operator.

\[
\text{qShiftAction(q: D, k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==}
\]
\[
q**(k*l)*\text{coefficient}(f, k)
\]

\[
\text{qShiftC(q: D, total: NonNegativeInteger): List D ==}
\]
\[
[q**i \text{ for i in } 0..\text{total}-1]
\]

-- Differentiation Operator

-- DiffAction(k, l, f) is the CoeffAction appropriate for the differentiation operator.

\[
\text{DiffAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==}
\]
\[
\text{coefficient}(f, (k-l)::\text{NonNegativeInteger})
\]

\[
\text{DiffC(total: NonNegativeInteger): List D ==}
\]
[0 for i in 1..total]

-- general - suitable for functions f

-- get the coefficient of z^k in the scalar product of p and f, the action
-- being defined by coeffAction

generalCoefficient(coeffAction: CoeffAction, f: Vector V,
   k: NonNegativeInteger, p: Vector SUP D): D ==
res: D := 0
for i in 1..#f repeat
  a := f.i
  b := p.i
  for l in minimumDegree b..degree b repeat
    if not zero? coefficient(b, l) then res := res + coefficient(b, l) * coeffAction(k, l, a)
res

generalInterpolation(C: List D, coeffAction: CoeffAction,
   f: Vector V,
   eta: List NonNegativeInteger): Matrix SUP D ==
  c: cFunction := (x,y) +-> generalCoefficient(coeffAction, f,
          (x-1)::NonNegativeInteger, y)
  fffg(C, c, eta)

-- general - suitable for functions f - trying all possible degree combinations

The following function returns the lexicographically next vector with non-negative components smaller than p with the same sum as v.

— package FFFG FractionFreeFastGaussian —

nextVector!(p: NonNegativeInteger, v: List NonNegativeInteger)
  : Union("failed", List NonNegativeInteger) ==
  n := #v
The following function returns the stream of all possible degree vectors, beginning with v, where the degree vectors are sorted in reverse lexicographic order. Furthermore, the entries are all less or equal to p and their sum equals the sum of the entries of v. We assume that the entries of v are also all less or equal to p.

--- package FFFG FractionFreeFastGaussian ---

vectorStream(p: NonNegativeInteger, v: List NonNegativeInteger) : Stream List NonNegativeInteger == delay
next := nextVector!(p, copy v)
(next case "failed") => empty()$Stream(List NonNegativeInteger)
cons(next, vectorStream(p, next))

---

vectorStream2 skips every second entry of vectorStream.

--- package FFFG FractionFreeFastGaussian ---

vectorStream2(p: NonNegativeInteger, v: List NonNegativeInteger) : Stream List NonNegativeInteger == delay
next := nextVector!(p, copy v)
(next case "failed") => empty()
next2 := nextVector!(p, copy next)
(next2 case "failed") => cons(next, empty())
cons(next2, vectorStream2(p, next2))

This version of generalInterpolation returns a stream of solutions, one for each possible
degree vector. Thus, it only needs to apply the previously defined generalInterpolation to
each degree vector. These are generated by vectorStream and vectorStream2 respectively.

If \( f \) consists of two elements only, we can skip every second degree vector: note that \( \text{fffg} \),
and thus also generalInterpolation, returns a matrix with \( \#f \) columns, each corresponding
to a solution of the interpolation problem. More precisely, the \( i \)th column is a solution with
degrees \( \eta - (1,1,\ldots,1) + e_i \). Thus, in the case of \( 2 \times 2 \) matrices, vectorStream would produce
solutions corresponding to \((d,0),(d-1,1),(d-2,2),(d-3,3),\ldots\), i.e.,
every second matrix is redundant.

Although some redundancy exists also for higher dimensional \( f \), the scheme becomes much
more complicated, thus we did not implement it.

— package FFFG FractionFreeFastGaussian —

\[
\text{generalInterpolation}(C: \text{List D}, \text{coeffAction: CoeffAction},
\text{f: Vector V},
\text{sumEta: NonNegativeInteger},
\text{maxEta: NonNegativeInteger})
: \text{Stream Matrix SUP D} ==
\]

\{
\text{generate an initial degree vector}\}

if \( \#f = 2 \) then
map(x +-> generalInterpolation(C, coeffAction, f, x),
cons(eta, vectorStream2(maxEta, eta)))
$\text{StreamFunctions2(\text{List NonNegativeInteger},}$
\text{Matrix SUP D})
else
map(x +-> generalInterpolation(C, coeffAction, f, x),
cons(eta, vectorStream(maxEta, eta)))
$\text{StreamFunctions2(\text{List NonNegativeInteger},}$
\text{Matrix SUP D})

We need to generate an initial degree vector, being the minimal element in reverse lexicographic order, i.e., \( m, m, \ldots, m, k, 0, 0, \ldots \), where \( m \) is \( \text{maxEta} \) and \( k \) is the remainder of
\( \text{sumEta} \) divided by \( \text{maxEta} \). This is done by the following code:

— generate an initial degree vector —
sum: Integer := sumEta
entry: Integer
eta: List NonNegativeInteger
:= [(if sum < maxEta 
then (entry := sum; sum := 0) 
else (entry := maxEta; sum := sum - maxEta); 
entry::NonNegativeInteger) for i in 1..#f]

We want to generate all vectors with sum of entries being at most sumEta. Therefore the following is incorrect.

--- BUG generate an initial degree vector ---

(\begin{itemize}
    \item (sum > 0) => empty($Stream$(Matrix SUP D))
\end{itemize})

--- package FFFG FractionFreeFastGaussian ---

Package FFFG FractionFreeFastGaussian

--- rational interpolation ---

\begin{verbatim}
interpolate(x: List Fraction D, y: List Fraction D, d: NonNegativeInteger) 
: Fraction SUP D ==
gx := splitDenominator(x)$InnerCommonDenominator(D, Fraction D, _
    List D, _
    List Fraction D)
gy := splitDenominator(y)$InnerCommonDenominator(D, Fraction D, _
    List D, _
    List Fraction D)
r := interpolate(gx.num, gy.num, d)
elt(numer r, monomial(gx.den,1))/(gy.den*elt(denom r, monomial(gx.den,1)))
\end{verbatim}

interpolate(x: List D, y: List D, d: NonNegativeInteger): Fraction SUP D ==
\begin{verbatim}
    -- berechne Interpolante mit Grad d und N-d-1
    if (N := #x) \neq #y then
        error "interpolate: number of points and values must match"
    if N <= d then
        error "interpolate: numerator degree must be smaller than number of data points"
    c: cFunction := (s,u) \to y.s * elt(u.2, x.s) - elt(u.1, x.s)
    eta: List NonNegativeInteger := [d, (N-d)::NonNegativeInteger]
    M := fffg(x, c, eta)
    if zero?(M.(2,1)) then M.(1,2)/M.(2,2)
    else M.(1,1)/M.(2,1)
\end{verbatim}
Because of Lemma 5.3, $M.1.(2,1)$ and $M.1.(2,2)$ cannot both vanish, since $\eta_{\sigma}$ is always $\sigma$-normal by Theorem 7.2 and therefore also para-normal, see Definition 4.2.

Because of Lemma 5.1 we have that $M.1.(*,2)$ is a solution of the interpolation problem, if $M.1.(2,1)$ vanishes.

--- package FFFG FractionFreeFastGaussian ---

recurrence computes the new matrix $M$, according to the following formulas (cf. Table 2 in Beckermann and Labahn):

\[
\begin{align*}
&\text{Increase order } \\
&\text{for } \ell = 1 \ldots m, \ell \neq \pi \\
&M^{(\cdot,\ell)}_{\sigma+1} := \left( M^{(\cdot,\ell)}_{\sigma}(\pi) - M^{(\cdot,\pi)}_{\sigma}(\ell) \right) / d_{\sigma}
\end{align*}
\]

\[
\begin{align*}
&\text{Increase order in column } \pi \\
&M^{(\cdot,\pi)}_{\sigma+1} := (z - c_{\sigma,\pi}) M^{(\cdot,\pi)}_{\sigma}
\end{align*}
\]

\[
\begin{align*}
&\text{Adjust degree constraints:} \\
&M^{(\cdot,\pi)}_{\sigma+1} := \left( M^{(\cdot,\pi)}_{\sigma+1}(\pi) - \sum_{\ell \neq \pi} M^{(\cdot,\ell)}_{\sigma+1}(\ell) \right) / d_{\sigma}
\end{align*}
\]

Since we do not need the matrix $M_{\sigma}$ anymore, we drop the index and update the matrix destructively. In the following, we write $C_k$ for $c_{\sigma,\pi}$.

--- package FFFG FractionFreeFastGaussian ---

-- a major part of the time is spent here

```
recurrence(M: Matrix SUP D, pi: NonNegativeInteger, m: NonNegativeInteger, 

  rPi: D := qelt(r, pi)
  polyf: SUP D := rPi * (z - Ck::SUP D)
  for i in 1..m repeat
    MiPi: SUP D := qelt(M, i, pi)
    newMiPi: SUP D := polyf * MiPi

  -- update columns ^= pi and calculate their sum
  for l in 1..m | l ^= pi repeat
    rl: D := qelt(r, l)
  -- I need the coercion to SUP D, since exquo returns an element of
```
-- Union("failed", SUP D)...
  Mil: SUP D := ((qelt(M, i, l) * rPi - MiPi * rl) exquo d)::SUP D
  qsetelt!(M, i, l, Mil)
  pl: D := qelt(p, l)
  newMiPi := newMiPi - pl * Mil

-- update column pi
  qsetelt!(M, i, pi, (newMiPi exquo d)::SUP D)

M

fffg(C: List D, c: cFunction, eta: List NonNegativeInteger): Matrix SUP D ==
-- eta is the vector of degrees. We compute M with degrees eta+e_i-1, i=1..m
  z: SUP D := monomial(1, 1)
  m: NonNegativeInteger := #eta
  M: Matrix SUP D := scalarMatrix(m, 1)
  d: D := 1
  K: NonNegativeInteger := reduce(_+, eta)
  etak: Vector NonNegativeInteger := zero(m)
  r: Vector D := zero(m)
  p: Vector D := zero(m)
  Lambda: List Integer
  lambdaMax: Integer
  lambda: NonNegativeInteger
  for k in 1..K repeat
    -- k = sigma+1
    for l in 1..m repeat r.l := c(k, column(M, l))
    Lambda := [eta.l-etak.l for l in 1..m | r.l ~= 0]
    -- if Lambda is empty, then M, d and etak remain unchanged. Otherwise, we look
    -- for the next closest para-normal point.
    (empty? Lambda) => "iterate"
    lambdaMax := reduce(max, Lambda)
    lambda := 1
    while eta.lambda-etak.lambda < lambdaMax or r.lambda = 0 repeat
      lambda := lambda + 1
      -- Calculate leading coefficients
      for l in 1..m | l ~= lambda repeat
        if etak.l > 0 then
          p.l := coefficient(M.(l, lambda),
             (etak.l-1)::NonNegativeInteger)
else
    p.l := 0

-- increase order and adjust degree constraints
M := recurrence(M, lambda, m, r, d, z, C.k, p)
d := r.lambda
etak.lambda := etak.lambda + 1
M

——

— FFFG.dotabb —
"FFFG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFG"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFG" -> "PFECAT"

——

package FFFGF FractionFreeFastGaussianFractions

—— FractionFreeFastGaussianFractions.input ——

)set break resume
)sys rm -f FractionFreeFastGaussianFractions.output
)spool FractionFreeFastGaussianFractions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionFreeFastGaussianFractions
--E 1

)spool
)lisp (bye)

——

— FractionFreeFastGaussianFractions.help ——
FractionFreeFastGaussianFractions examples

This package lifts the interpolation functions from
FractionFreeFastGaussian to fractions.

The packages defined in this file provide fast fraction free rational
interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

See Also:
o )show FractionFreeFastGaussianFractions

FractionFreeFastGaussianFractions (FFFGF)

Exports:
generalInterpolation

--- package FFFGF FractionFreeFastGaussianFractions ---

)abbrev package FFFGF FractionFreeFastGaussianFractions
++ Author: Martin Rubey
++ Description:
++ This package lifts the interpolation functions from
++ \spadtype{FractionFreeFastGaussian} to fractions.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussianFractions(D, V, VF): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: FiniteAbelianMonoidRing(D, NonNegativeInteger)
VF: FiniteAbelianMonoidRing(Fraction D, NonNegativeInteger)

F  ==> Fraction D
NNI ==> NonNegativeInteger
SUP ==> SparseUnivariatePolynomial
FFFG ==> FractionFreeFastGaussian
FAMR2 ==> FiniteAbelianMonoidRingFunctions2

cFunction ==> (NNI, Vector SUP D) -> D

CoeffAction ==> (NNI, NNI, V) -> D
-- coeffAction(k, l, f) is the coefficient of x^k in z^l f(x)

Exports == with

generalInterpolation: (List D, CoeffAction, Vector VF, List NNI)
  -> Matrix SUP D
++ \spad{generalInterpolation(l, CA, f, eta)} performs Hermite-Pade
++ approximation using the given action CA of polynomials on the elements
++ of f. The result is guaranteed to be correct up to order
++ |eta|-1. Given that eta is a "normal" point, the degrees on the
++ diagonal are given by eta. The degrees of column i are in this case
++ eta + e.i - [1,1,...,1], where the degree of zero is -1.

generalInterpolation: (List D, CoeffAction, Vector VF, NNI, NNI)
  -> Stream Matrix SUP D
++ \spad{generalInterpolation(l, CA, f, sumEta, maxEta)} applies
++ generalInterpolation(l, CA, f, eta) for all possible eta with maximal
++ entry maxEta and sum of entries sumEta

Implementation == add

multiplyRows!(v: Vector D, M: Matrix SUP D): Matrix SUP D ==
n := #v
for i in 1..n repeat
  for j in 1..n repeat
    M.(i,j) := v.i*M.(i,j)
  end for

M

generalInterpolation(C: List D, coeffAction: CoeffAction,
  f: Vector VF, eta: List NNI): Matrix SUP D ==
n := #f
g: Vector V := new(n, 0)
den: Vector D := new(n, 0)

for i in 1..n repeat
  c := coefficients(f.i)
den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
g.i :=
map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)

M := generalInterpolation(C, coeffAction, g, eta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not
-- each column. Possibly I could factor out gcd den, but I'm not sure whether
-- this is efficient.
multiplyRows!(den, M)

generalInterpolation(C: List D, coeffAction: CoeffAction,
f: Vector VF, sumEta: NNI, maxEta: NNI)
: Stream Matrix SUP D ==

n := #f
g: Vector V := new(n, 0)
den: Vector D := new(n, 0)

for i in 1..n repeat
c := coefficients(f.i)
den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
g.i :=
map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)
c: cFunction :=
(x,y) +-> generalCoefficient(coeffAction, g, (x-1)::NNI, y)$FFFG(D, V)

MS: Stream Matrix SUP D
:= generalInterpolation(C, coeffAction, g, sumEta, maxEta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not
-- each column. Possibly I could factor out gcd den, but I'm not sure whether
-- this is efficient.
map(x +-> multiplyRows!(den, x), MS)$Stream(Matrix SUP D)

— FFFGF.dotabb —
"FFFGF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFGF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFGF" -> "PFECAT"
package FRAC2 FractionFunctions2

--- FractionFunctions2.input ---

)set break resume
)sys rm -f FractionFunctions2.output
)spool FractionFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FractionFunctions2
--E 1

)spool
)lisp (bye)

---

--- FractionFunctions2.help ---

====================================================================
FractionFunctions2 examples
====================================================================

This package extends a map between integral domains to a map between
Fractions over those domains by applying the map to the numerators and
denominators.

See Also:
o )show FractionFunctions2

---
FractionFunctions2 (FRAC2)

Exports:
map

— package FRAC2 FractionFunctions2 —

)abbrev package FRAC2 FractionFunctions2
++ Description:
++ This package extends a map between integral domains to
++ a map between Fractions over those domains by applying the map to the
++ numerators and denominators.

FractionFunctions2(A, B): Exports == Impl where
  A, B: IntegralDomain
  R ==> Fraction A
  S ==> Fraction B

Exports ==> with
map: (A -> B, R) -> S
  ++ map(func,frac) applies the function func to the numerator
  ++ and denominator of the fraction frac.

Impl ==> add
  map(f, r) == map(f, r)$QuotientFieldCategoryFunctions2(A, B, R, S)
package FRNAAF2 FramedNonAssociativeAlgebraFunctions2

— FramedNonAssociativeAlgebraFunctions2.input —

)set break resume
)sys rm -f FramedNonAssociativeAlgebraFunctions2.output
)spool FramedNonAssociativeAlgebraFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FramedNonAssociativeAlgebraFunctions2

--E 1

)spool
)lisp (bye)

— FramedNonAssociativeAlgebraFunctions2.help —

====================================================================
FramedNonAssociativeAlgebraFunctions2 examples
====================================================================

FramedNonAssociativeAlgebraFunctions2 implements functions between
two framed non associative algebra domains defined over different rings.
The function map is used to coerce between algebras over different
domains having the same structural constants.

See Also:
o )show FramedNonAssociativeAlgebraFunctions2
Exports:
map

— package FRNAAF2 FramedNonAssociativeAlgebraFunctions2 —

)abbrev package FRNAAF2 FramedNonAssociativeAlgebraFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 28 February 1992
++ Date Last Updated: 28 February 1992
++ Description:
++ FramedNonAssociativeAlgebraFunctions2 implements functions between
++ two framed non associative algebra domains defined over different rings.
++ The function map is used to coerce between algebras over different
++ domains having the same structural constants.

FramedNonAssociativeAlgebraFunctions2(AR,R,AS,S) : Exports ==
Implementation where
R : CommutativeRing
S : CommutativeRing
AR : FramedNonAssociativeAlgebra R
AS : FramedNonAssociativeAlgebra S
V ==> Vector
Exports ==> with
map: (R -> S, AR) -> AS
++ map(f,u) maps f onto the coordinates of u to get an element
++ in \spad{AS} via identification of the basis of \spad{AR}
++ as beginning part of the basis of \spad{AS}.
Implementation ==> add
map(fn : R -> S, u : AR) : AS ==
rank($)AR > rank($)AS => error("map: ranks of algebras do not fit")
vr : V R := coordinates u
vs : V S := map(fn,vr)$VectorFunctions2(R,S)

|||-
This line used to read:

\[ \text{rank()$AR = rank()$AR \Rightarrow represents(vs)$AS} \]

but the test is clearly always true and cannot be what was intended. Gregory Vanuxem supplied the fix below.

--- package FRNAAF2 FramedNonAssociativeAlgebraFunctions2 ---

\[
\text{rank()$AR = rank()$AS \Rightarrow represents(vs)$AS} \\
\text{ba := basis()$AS} \\
\text{represents(vs,[ba.i for i in 1..rank()$AR])}
\]

---

--- FRNAAF2.dotabb ---

"FRNAAF2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRNAAF2"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"FRNAAF2" -> "FRNAALG"

---

package FSPECF FunctionalSpecialFunction

--- FunctionalSpecialFunction.input ---

)set break resume
)sys rm -f FunctionalSpecialFunction.output
)spool FunctionalSpecialFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionalSpecialFunction
--E 1

)spool
)lisp (bye)

--- FunctionalSpecialFunction.help ---
FunctionalSpecialFunction examples

Provides some special functions over an integral domain.

See Also:
- )show FunctionalSpecialFunction

---

FunctionalSpecialFunction (FSPECF)

Exports:
- Beta
- Gamma
- abs
- airyAi
- airyBi
- belong?
- besselI
- besselJ
- besselK
- besselY
- digamma
- iiAiryAi
- iiAiryBi
- iiBesselI
- iiBesselJ
- iiBesselK
- iiBesselY
- iiBeta
- iiGamma
- liabs
- iidigamma
- iipolygamma
- operator
- polygamma

--- package FSPECF FunctionalSpecialFunction ---

)abbrev package FSPECF FunctionalSpecialFunction
++ Author: Manuel Bronstein
++ Date Created: 18 Apr 1989
++ Date Last Updated: 4 October 1993
++ Description:
++ Provides some special functions over an integral domain.

FunctionalSpecialFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R
  OP  => BasicOperator
K ==> Kernel F
SE ==> Symbol
SPECIALDIFF ==> "%specialDiff"

Exports ==> with
  belong? : OP -> Boolean
    ++ belong?(op) is true if op is a special function operator;
  operator: OP -> OP
    ++ operator(op) returns a copy of op with the domain-dependent
    ++ properties appropriate for F;
    ++ error if op is not a special function operator
  abs : F -> F
    ++ abs(f) returns the absolute value operator applied to f
  Gamma : F -> F
    ++ Gamma(f) returns the formal Gamma function applied to f
  Gamma : (F,F) -> F
    ++ Gamma(a,x) returns the incomplete Gamma function applied to a and x
  Beta: (F,F) -> F
    ++ Beta(x,y) returns the beta function applied to x and y
  digamma: F->F
    ++ digamma(x) returns the digamma function applied to x
  polygamma: (F,F) ->F
    ++ polygamma(x,y) returns the polygamma function applied to x and y
  besselJ: (F,F) -> F
    ++ besselJ(x,y) returns the besselj function applied to x and y
  besselY: (F,F) -> F
    ++ besselY(x,y) returns the bessely function applied to x and y
  besselI: (F,F) -> F
    ++ besselI(x,y) returns the besseli function applied to x and y
  besselK: (F,F) -> F
    ++ besselK(x,y) returns the besselk function applied to x and y
  airyAi: F -> F
    ++ airyAi(x) returns the airyai function applied to x
  airyBi: F -> F
    ++ airyBi(x) returns the airybi function applied to x

In case we want to have more special function operators here, do not forget to add them to
the list specop in CommonOperators. Otherwise they will not have the 'special' attribute
and will not be recognised here. One effect could be that

myNewSpecOp(1::Expression Integer)::Expression DoubleFloat

might not re-evaluate the operator.

    — package FSPECF FunctionalSpecialFunction —

    iiGamma : F -> F
+++ iiGamma(x) should be local but conditional;
iabs : F -> F
+++ iiabs(x) should be local but conditional;
iBeta : List F -> F
+++ iiBeta(x) should be local but conditional;
iidigamma : F -> F
+++ iidigamma(x) should be local but conditional;
iipolygamma : List F -> F
+++ iipolygamma(x) should be local but conditional;
iBesselJ : List F -> F
+++ iiBesselJ(x) should be local but conditional;
iBesselY : List F -> F
+++ iiBesselY(x) should be local but conditional;
iBesselI : List F -> F
+++ iiBesselI(x) should be local but conditional;
iBesselK : List F -> F
+++ iiBesselK(x) should be local but conditional;
iiAiryAi : F -> F
+++ iiAiryAi(x) should be local but conditional;
iiAiryBi : F -> F
+++ iiAiryBi(x) should be local but conditional;

Implementation ==> add

iabs : F -> F
iGamma : F -> F
iBeta : (F, F) -> F
idigamma : F -> F
iipolygamma : (F, F) -> F
iiBesselJ : (F, F) -> F
iiBesselY : (F, F) -> F
iiBesselI : (F, F) -> F
iiBesselK : (F, F) -> F
iiAiryAi : F -> F
iiAiryBi : F -> F

opabs := operator("abs"::Symbol)$CommonOperators
opGamma := operator("Gamma"::Symbol)$CommonOperators
opBeta := operator("Beta"::Symbol)$CommonOperators
opdigamma := operator("digamma"::Symbol)$CommonOperators
oppolygamma := operator("polygamma"::Symbol)$CommonOperators
opBesselJ := operator("besselJ"::Symbol)$CommonOperators
opBesselY := operator("besselY"::Symbol)$CommonOperators
opBesselI := operator("besselI"::Symbol)$CommonOperators
opBesselK := operator("besselK"::Symbol)$CommonOperators
opAiryAi := operator("airyAi"::Symbol)$CommonOperators
opAiryBi := operator("airyBi"::Symbol)$CommonOperators

abs x == opabs x
Gamma(x) == opGamma(x)
\[
\begin{align*}
\text{Gamma}(a, x) &= \text{opGamma2}(a, x) \\
\text{Beta}(x, y) &= \text{opBeta}(x, y) \\
\text{digamma} \ x &= \text{opdigamma}(x) \\
\text{polygamma}(k, x) &= \text{oppolygamma}(k, x) \\
\text{besselJ}(a, x) &= \text{opBesselJ}(a, x) \\
\text{besselY}(a, x) &= \text{opBesselY}(a, x) \\
\text{besselI}(a, x) &= \text{opBesselI}(a, x) \\
\text{besselK}(a, x) &= \text{opBesselK}(a, x) \\
\text{airyAi}(x) &= \text{opAiryAi}(x) \\
\text{airyBi}(x) &= \text{opAiryBi}(x)
\end{align*}
\]

\text{belong? op} \ = \ \text{has?(op, "special")}

\text{operator op} =
\begin{align*}
\text{is?(op, "abs"::Symbol)} & => \text{opabs} \\
\text{is?(op, "Gamma"::Symbol)} & => \text{opGamma} \\
\text{is?(op, "Gamma2"::Symbol)} & => \text{opGamma2} \\
\text{is?(op, "Beta"::Symbol)} & => \text{opBeta} \\
\text{is?(op, "digamma"::Symbol)} & => \text{opdigamma} \\
\text{is?(op, "polygamma"::Symbol)} & => \text{oppolygamma} \\
\text{is?(op, "besselJ"::Symbol)} & => \text{opBesselJ} \\
\text{is?(op, "besselY"::Symbol)} & => \text{opBesselY} \\
\text{is?(op, "besselI"::Symbol)} & => \text{opBesselI} \\
\text{is?(op, "besselK"::Symbol)} & => \text{opBesselK} \\
\text{is?(op, "airyAi"::Symbol)} & => \text{opAiryAi} \\
\text{is?(op, "airyBi"::Symbol)} & => \text{opAiryBi}
\end{align*}

\text{error "Not a special operator"}

\text{-- Could put more unconditional special rules for other functions here}
\text{iGamma x} =
\begin{align*}
\text{one? \ x} & => \ x \\
(x = 1) & => \ x \\
\text{kernel}(\text{opGamma}, \ x)
\end{align*}

\text{iabs x} =
\begin{align*}
\text{zero? \ x} & => 0 \\
\text{is?(x, opabs)} & => \ x \\
x < 0 & => \text{kernel}(\text{opabs}, \ -x) \\
\text{kernel}(\text{opabs}, \ x)
\end{align*}

\text{iBeta(x, y)} = \text{kernel}(\text{opBeta}, [x, y])
\text{idigamma x} = \text{kernel}(\text{opdigamma}, x)
\text{iiipolygamma(n, x)} = \text{kernel}(\text{oppolygamma}, [n, x])
\text{iiibesselJ(x, y)} = \text{kernel}(\text{opBesselJ}, [x, y])
\text{iiibesselY(x, y)} = \text{kernel}(\text{opBesselY}, [x, y])
\text{iiibesselI(x, y)} = \text{kernel}(\text{opBesselI}, [x, y])
\text{iiibesselK(x, y)} = \text{kernel}(\text{opBesselK}, [x, y])
\text{iAiryAi x} = \text{kernel}(\text{opAiryAi}, x)
\text{iAiryBi x} = \text{kernel}(\text{opAiryBi}, x)
-- Could put more conditional special rules for other functions here

if R has abs : R -> R then
  iiabs x ==
  (r := retractIfCan(x)@Union(Fraction Polynomial R, "failed"))
  case "failed" => iiabs x
  f := r::Fraction Polynomial R
  (a := retractIfCan(numer f)@Union(R, "failed")) case "failed" or
  (b := retractIfCan(denom f)@Union(R,"failed")) case "failed" => iiabs x
  abs(a::R)::F / abs(b::R)::F
else iiabs x == iiabs x

if R has SpecialFunctionCategory then
  iiGamma x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iiGamma x
  Gamma(r::R)::F

  iiBeta l ==
  (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" or
  => iiBeta(first l, second l)
  Beta(r::R, s::R)::F

  iidigamma x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iidigamma x
  digamma(r::R)::F

  iiipolygamma l ==
  (s:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (r:=retractIfCan(second l)@Union(R,"failed")) case "failed" or
  => iiipolygamma(first l, second l)
  polygamma(s::R, r::R)::F

  iiBesselJ l ==
  (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" or
  => iiBesselJ(first l, second l)
  besselJ(r::R, s::R)::F

  iiBesselY l ==
  (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" or
  => iiBesselY(first l, second l)
  besselY(r::R, s::R)::F

  iiBesselI l ==
  (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" or
  => iiBesselI(first l, second l)
  besselI(r::R, s::R)::F
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" => iiiBesselI(first l, second l)
besselI(r::R, s::R)::F

iiBesselK l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
    => iiiBesselK(first l, second l)
besselK(r::R, s::R)::F

iiAiryAi x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryAi x
airyAi(r::R)::F

iiAiryBi x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryBi x
airyBi(r::R)::F

else
    if R has RetractableTo Integer then
        iiGamma x ==
            (r := retractIfCan(x)@Union(Integer, "failed")) case Integer
            and (r::Integer >= 1) => factorial(r::Integer - 1)::F
        iGamma x
    else
        iiGamma x == iGamma x

iiBeta l == iBeta(first l, second l)
iidigamma x == idigamma x
iiPolygamma l == iiPolygamma(first l, second l)
iiBesselJ l == iiiBesselJ(first l, second l)
iiBesselY l == iiiBesselY(first l, second l)
iiBesselI l == iiiBesselI(first l, second l)
iiBesselK l == iiiBesselK(first l, second l)
iiAiryAi x == iAiryAi x
iiAiryBi x == iAiryBi x

-- Default behaviour is to build a kernel
evaluate(opGamma, iiGamma)$BasicOperatorFunctions1(F)
evaluate(opabs, iiabs)$BasicOperatorFunctions1(F)
evaluate(opdigamma, iidigamma)$BasicOperatorFunctions1(F)
evaluate(oppolygamma, iiPolygamma)$BasicOperatorFunctions1(F)
evaluate(opBesselJ, iiBesselJ)$BasicOperatorFunctions1(F)
evaluate(opBesselY, iiBesselY)$BasicOperatorFunctions1(F)
evaluate(opBesselI, iiBesselI)$BasicOperatorFunctions1(F)
evaluate(opBesselK, iiBesselK)$BasicOperatorFunctions1(F)
evaluate(opAiryAi, iiAiryAi)$BasicOperatorFunctions1(F)
evaluate(opAiryBi, iiAiryBi)$BasicOperatorFunctions1(F)
differentiation of special functions

In the following we define the symbolic derivatives of the special functions we provide. The formulas we use for the Bessel functions can be found in Milton Abramowitz and Irene A. Stegun, eds. (1965). Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover. ISBN 0-486-61272-4, Equations 9.1.27 and 9.6.26. Up to patch-50 the formula for $K$ missed the minus sign. (Issue #355)

We do not attempt to provide formulas for the derivative with respect to the first argument currently. Instead, we leave such derivatives unevaluated.

---

```plaintext
import Fraction Integer
ahalf: F := recip(2::F)::F
athird: F := recip(2::F)::F
twothirds: F := 2*recip(3::F)::F
```

---

We need to get hold of the differentiation operator as modified by FunctionSpace. Otherwise, for example, display will be ugly. We accomplish that by differentiating an operator, which will certainly result in a single kernel only.

---

```plaintext
dummyArg: SE := new()$SE
opdiff := operator first kernels D((operator(new()$SE)$BasicOperator)
                                  (dummyArg::F), dummyArg)
```

---

The differentiation operator opdiff takes three arguments corresponding to

$F_{i,i}(a_1, a_2, \ldots, a_n)$:

1. $F(a_1, \ldots, dm, \ldots a_n)$, where the $i$\textsuperscript{th} argument is a dummy variable,
2. $dm$, the dummy variable, and
3. $a_i$, the point at which the differential is evaluated.

In the following, it seems to be safe to use the same dummy variable throughout. At least, this is done also in FunctionSpace, and did not cause problems.

The operation symbolicGrad returns the first component of the gradient of op l.
— package FSPECFF FunctionalSpecialFunction —

dm := new()$SE :: F

iBesselJ(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t)*kernel(opdiff, [opBesselJ [dm, x], dm, n])
+ differentiate(x, t) * ahalf * (besselJ (n-1,x) - besselJ (n+1,x))

iBesselY(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t)*kernel(opdiff, [opBesselY [dm, x], dm, n])
+ differentiate(x, t) * ahalf * (besselY (n-1,x) - besselY (n+1,x))

iBesselI(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t)*kernel(opdiff, [opBesselI [dm, x], dm, n])
+ differentiate(x, t)* ahalf * (besselI (n-1,x) + besselI (n+1,x))

iBesselK(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t)*kernel(opdiff, [opBesselK [dm, x], dm, n])
- differentiate(x, t)* ahalf * (besselK (n-1,x) + besselK (n+1,x))

———

For the moment we throw an error if we try to differentiate polygamma with respect to the first argument.

— package FSPECFF FunctionalSpecialFunction —

ipolygamma(l: List F, x: SE): F ==
member?(x, variables first l) =>
  error "cannot differentiate polygamma with respect to the first argument"
n := first l; y := second l
differentiate(y, x)*polygamma(n+1, y)

iBetaGrad1(l: List F): F ==
x := first l; y := second l
Beta(x,y)*(digamma x - digamma(x+y))

iBetaGrad2(l: List F): F ==
x := first l; y := second l
Beta(x,y)*(digamma y - digamma(x+y))

if F has ElementaryFunctionCategory then
  iGamma2(l: List F, t: SE): F ==
a := first l; x := second l
differentiate(a, t)*kernel(opdiff, [opGamma2 [dm, x], dm, a])
Finally, we tell Axiom to use these functions for differentiation. Note that up to patch 50, the properties for the Bessel functions were set using derivative(oppolygamma, [lzero, ipolygammaGrad]), where lzero returned zero always. Trying to replace lzero by a function that returns the first component of the gradient failed, it resulted in an infinite loop for integrate(D(besselJ(a,x),a,a)).

— package FSPECF FunctionalSpecialFunction —

derivative(opabs, (x:F):F +-> abs(x) * inv(x))
derivative(opGamma, (x:F):F +-> digamma x * Gamma x)
derivative(opBeta, [iBetaGrad1, iBetaGrad2])
derivative(opdigamma, (x:F):F +-> polygamma(1, x))
setProperty(oppolygamma, SPECIALDIFF, ipolygamma@((List F, SE)->F)
pretend None)
setProperty(opBesselJ, SPECIALDIFF, iBesselJ@((List F, SE)->F)
pretend None)
setProperty(opBesselY, SPECIALDIFF, iBesselY@((List F, SE)->F)
pretend None)
setProperty(opBesselI, SPECIALDIFF, iBesselI@((List F, SE)->F)
pretend None)
setProperty(opBesselK, SPECIALDIFF, iBesselK@((List F, SE)->F)
pretend None)

— FSPECF.dotabb —

"FSPECF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSPECF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSPECF" -> "FS"
FunctionFieldCategoryFunctions2 (FFCAT2)

Exports:

map

Lifts a map from rings to function fields over them.

See Also:

o )show FunctionFieldCategoryFunctions2

---

FunctionFieldCategoryFunctions2 (FFCAT2) examples

Lifts a map from rings to function fields over them.

See Also:

o )show FunctionFieldCategoryFunctions2

---

--- FunctionFieldCategoryFunctions2.help ---

Lifts a map from rings to function fields over them.

See Also:

o )show FunctionFieldCategoryFunctions2

---

Exports:

map

---
— package FFCAT2 FunctionFieldCategoryFunctions2 —

)abbrev package FFCAT2 FunctionFieldCategoryFunctions2
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 26 Jul 1988
++ Description:
++ Lifts a map from rings to function fields over them.

FunctionFieldCategoryFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):
Exports == Implementation where
  R1 : UniqueFactorizationDomain
  UP1 : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  F1 : FunctionFieldCategory(R1, UP1, UPUP1)
  R2 : UniqueFactorizationDomain
  UP2 : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2
  F2 : FunctionFieldCategory(R2, UP2, UPUP2)

Exports ==> with
  map: (R1 -> R2, F1) -> F2
    ++ map(f, p) lifts f to F1 and applies it to p.

Implementation ==> add
  map(f, f1) ==
    reduce(map(f, lift f1)$MultipleMap(R1, UP1, UPUP1, R2, UP2, UPUP2))

— FFCAT2.dotabb —

"FFCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFCAT2"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FFCAT2" -> "FFCAT"

——

package FFINTBAS FunctionFieldIntegralBasis

— FunctionFieldIntegralBasis.input —

)set break resume
Integral bases for function fields of dimension one

In this package \( R \) is a Euclidean domain and \( F \) is a framed algebra over \( R \). The package provides functions to compute the integral closure of \( R \) in the quotient field of \( F \). It is assumed that \( \text{char}(R/P) = \text{char}(R) \) for any prime \( P \) of \( R \). A typical instance of this is when \( R = K[x] \) and \( F \) is a function field over \( R \).

See Also:
- \( \)show FunctionFieldIntegralBasis

---

**FunctionFieldIntegralBasis (FFINTBAS)**
Exports:
  integralBasis  localIntegralBasis

— package FFINTBAS FunctionFieldIntegralBasis —

)abbrev package FFINTBAS FunctionFieldIntegralBasis
++ Author: Victor Miller
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Description:
++ Integral bases for function fields of dimension one
++ In this package R is a Euclidean domain and F is a framed algebra
++ over R. The package provides functions to compute the integral
++ closure of R in the quotient field of F. It is assumed that
++ \spad{\text{char}(R/P) = \text{char}(R)} for any prime P of R. A typical instance of
++ this is when \spad{R = K[x]} and F is a function field over R.

FunctionFieldIntegralBasis(R,UP,F): Exports == Implementation where
  R : EuclideanDomain with
    squareFree: $ \rightarrow $ Factored $
    ++ \text{squareFree}(x) $ returns a square-free factorisation of x
  UP : UnivariatePolynomialCategory R
  F : FramedAlgebra(R,UP)

I  ==> Integer
Mat ==> Matrix R
NNI ==> NonNegativeInteger

Exports == with
  integralBasis : () \rightarrow \text{Record(basis: Mat, basisDen: R, basisInv:Mat)}
    ++ \spad{\text{integralBasis}()} returns a record
    ++ \spad{\text{integralBasis}(\text{basis})} containing information regarding
    ++ the integral closure of R in the quotient field of F, where
    ++ F is a framed algebra with R-module basis \spad{\text{basis}(w_1,w_2,\ldots,w_n)}.
    ++ If \spad{\text{basis}(i)} is the matrix \spad{\text{basis}((a_{ij}, i = 1..n, j = 1..n))}, then
    ++ the \spad{i}th element of the integral basis is
    ++ \spad{\text{basis}(i) = (1/\text{basisDen}) \ast \text{sum}(a_{ij} \ast w_j, j = 1..n)}, i.e. the
    ++ \spad{i}th row of \spad{\text{basis}} contains the coordinates of the
    ++ \spad{i}th basis vector. Similarly, the \spad{i}th row of the
    ++ matrix \spad{\text{basisInv}(i)} contains the coordinates of \spad{\text{basis}(w_i)} with
    ++ respect to the basis \spad{\text{basis}(v_1,\ldots,v_n)}: if \spad{\text{basisInv}(i)} is the
    ++ matrix \spad{\text{basisInv}((b_{ij}, i = 1..n, j = 1..n))}, then
    ++ \spad{\text{basisInv}(i) = \text{sum}(b_{ij} \ast v_j, j = 1..n)}.
  localIntegralBasis : R \rightarrow \text{Record(basis: Mat, basisDen: R, basisInv:Mat)}
    ++ \spad{\text{localIntegralBasis}(p)} returns a record
    ++ \spad{\text{localIntegralBasis}(p)} containing information regarding
    ++ the local integral closure of R at the prime \spad{p} in the quotient
    ++ field of F, where F is a framed algebra with R-module basis
\texttt{\textbackslash spad\{w1, w2, \ldots, wn\}.}
+ If \texttt{\textbackslash spad\{basis\}} is the matrix \texttt{\textbackslash spad\{(aij, i = 1..n, j = 1..n)\}}, then
+ the \texttt{\textbackslash spad\{i\}th element of the local integral basis is}
+ \texttt{\textbackslash spad\{vi = (1/basisDen) \ast \text{sum(aij \ast wj, j = 1..n)}\}}, i.e. the
+ \texttt{\textbackslash spad\{i\}th row of \texttt{\textbackslash spad\{basis\}} contains the coordinates of the
+ \texttt{\textbackslash spad\{i\}th basis vector. Similarly, the \texttt{\textbackslash spad\{i\}th row of the
+ \texttt{\textbackslash spad\{basisInv\}} contains the coordinates of \texttt{\textbackslash spad\{wi\}} with
+ respect to the basis \texttt{\textbackslash spad\{v1, \ldots, vn\}}: if \texttt{\textbackslash spad\{basisInv\}} is the
+ \texttt{\textbackslash spad\{bi, i = 1..n, j = 1..n\}}, then
+ \texttt{\textbackslash spad\{wi = \text{sum(bij \ast vj, j = 1..n)}\}.}

Implementation \Rightarrow add

\begin{verbatim}
import IntegralBasisTools(R, UP, F)
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

squaredFactors: R -> R
squaredFactors px ==
  /*[(if ffe.exponent > 1 then ffe.factor else 1$R)
   for ffe in factors squareFree px]

iIntegralBasis: (Mat, R, R) -> Record(basis: Mat, basisDen: R, basisInv: Mat)
iIntegralBasis(tfm, disc, sing) ==
  -- tfm = trace matrix of current order
  n := rank()$F; tfm0 := copy tfm; disc0 := disc
  rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
  -- rb = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  rbden : R := 1; index : R := 1; oldIndex : R := 1
  -- rbden = denominator for current basis matrix
  -- index = index of original order in current order
  not sizeLess?(1, sing) => [rb, rbden, rbinv]
  repeat
  -- compute the p-radical
  idinv := transpose squareTop rowEchelon(tfm, sing)
  -- [u1, \ldots, un] are the coordinates of an element of the p-radical
  -- iff [u1, \ldots, un] \ast \text{indiv} is in sing \ast R^n
  id := rowEchelon LowTriBddDenomInv(idinv, sing)
  -- id = basis matrix of the p-radical
  idinv := UpTriBddDenomInv(id, sing)
  -- id \ast idinv = sing \ast identity
  -- no need to check for inseparability in this case
  rbinv := idealiser(id \ast rb, rbinv \ast idinv, sing \ast rbden)
  index := diagonalProduct rbinv
  rb := rowEchelon LowTriBddDenomInv(rbinv, rbden \ast sing)
  g := matrixGcd(rb, sing, n)
  if sizeLess?(1, g) then rb := (rb exquo g) :: Mat
  rbden := rbden \ast (sing quo g)
  rbinv := UpTriBddDenomInv(rb, rbden)
\end{verbatim}
disc := disc0 quo (index * index)
indexChange := index quo oldIndex; oldIndex := index
sing := gcd(indexChange, squaredFactors disc)
not sizeLess?(1, sing) => return [rb, rbden, rbinv]
tfm := ((rb * tfm0 * transpose rb) exquo (rbden * rbden)) :: Mat

integralBasis() ==
  n := rank()$F; p := characteristic()$F
  (not zero? p) and (n >= p) =>
    error "integralBasis: possible wild ramification"
tfm := traceMatrix()$F; disc := determinant tfm
sing := squaredFactors disc -- singularities of relative Spec
iIntegralBasis(tfm,disc,sing)

localIntegralBasis prime ==
  n := rank()$F; p := characteristic()$F
  (not zero? p) and (n >= p) =>
    error "integralBasis: possible wild ramification"
tfm := traceMatrix()$F; disc := determinant tfm
  (disc exquo (prime * prime)) case "failed" =>
    [scalarMatrix(n,1),1,scalarMatrix(n,1)]
iIntegralBasis(tfm,disc,prime)

———

— FFINTBAS.dotabb —

"FFINTBAS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFINTBAS"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FFINTBAS" -> "FRAMALG"

———

package PMASSFS FunctionSpaceAssertions

— FunctionSpaceAssertions.input —

)set break resume
)sys rm -f FunctionSpaceAssertions.output
)spool FunctionSpaceAssertions.output
)set message test on
)set message auto off
)clear all
FunctionSpaceAssertions examples

Attaching assertions to symbols for pattern matching;

See Also:
o  )show FunctionSpaceAssertions

Exports:
assert constant multiple optional

— package PMASSFS FunctionSpaceAssertions —

)abbrev package PMASSFS FunctionSpaceAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching assertions to symbols for pattern matching;

FunctionSpace Assertions (R, F): Exports == Implementation where
R: Ordered Set
F: Function Space R

K ==> Kernel F
PMOPT ==> "%pmoptional"
PMMULT ==> "%pmmultiple"
PMCONST ==> "%pmconstant"

Exports ==> with
    assert : (F, String) -> F
        ++ assert(x, s) makes the assertion s about x.
        ++ Error: if x is not a symbol.
    constant: F -> F
        ++ constant(x) tells the pattern matcher that x should
        ++ match only the symbol 'x and no other quantity.
        ++ Error: if x is not a symbol.
    optional: F -> F
        ++ optional(x) tells the pattern matcher that x can match
        ++ an identity (0 in a sum, 1 in a product or exponentiation).
        ++ Error: if x is not a symbol.
    multiple: F -> F
        ++ multiple(x) tells the pattern matcher that x should
        ++ preferably match a multi-term quantity in a sum or product.
        ++ For matching on lists, multiple(x) tells the pattern matcher
        ++ that x should match a list instead of an element of a list.
        ++ Error: if x is not a symbol.

Implementation ==> add
    ass : (K, String) -> F
    asst : (K, String) -> F
    mkk : Basic Operator -> F

    mkk op == kernel(op, empty()$List(F))

    ass(k, s) ==
        has?(op := operator k, s) => k::F
        mkk assert(copy op, s)

    asst(k, s) ==
        has?(op := operator k, s) => k::F
        mkk assert(op, s)

    assert(x, s) ==
        retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
            asst(retract(x)@K, s)
error "assert must be applied to symbols only"

constant x ==
  retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
  ass(retract(x)@K, PMCONST)
  error "constant must be applied to symbols only"

optional x ==
  retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
  ass(retract(x)@K, PMOPT)
  error "optional must be applied to symbols only"

multiple x ==
  retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
  ass(retract(x)@K, PMMULT)
  error "multiple must be applied to symbols only"

— PMASSFS.dotabb —

"PMASSFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMASSFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMASSFS" -> "FS"

— package PMPREDFS FunctionSpaceAttachPredicates —

package PMPREDFS FunctionSpaceAttachPredicates

   — FunctionSpaceAttachPredicates.input —

)set break resume
)sys rm -f FunctionSpaceAttachPredicates.output
)spool FunctionSpaceAttachPredicates.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceAttachPredicates
--E 1

)spool
)lisp (bye)
Attaching predicates to symbols for pattern matching.

See Also:
o )show FunctionSpaceAttachPredicates

Exports:
suchThat

---
package PMPREDFS FunctionSpaceAttachPredicates ---

)abbrev package PMPREDFS FunctionSpaceAttachPredicates
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching predicates to symbols for pattern matching.

FunctionSpaceAttachPredicates(R, F, D): Exports == Implementation where
  R: OrderedSet
  F: FunctionSpace R
  D: Type
K ==> Kernel F
PMPRED ==> "%pmpredicate"

Exports ==> with
  suchThat: (F, D -> Boolean) -> F
  ++ suchThat(x, foo) attaches the predicate foo to x;
  ++ error if x is not a symbol.
suchThat: (F, List(D -> Boolean)) -> F
  ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
  ++ f1 and f2 and ... and fn to x.
  ++ Error: if x is not a symbol.

Implementation ==> add
import AnyFunctions1(D -> Boolean)

st : (K, List Any) -> F
preds: K -> List Any
mkk : BasicOperator -> F

suchThat(p:F, f:D -> Boolean) == suchThat(p, [f])
mkk op == kernel(op, empty()$List(F))

preds k ==
  (u := property(operator k, PMPRED)) case "failed" => empty()
  (u::None) pretend List(Any)

st(k, l) ==
  mkk assert(setProperty(copy operator k, PMPRED,
                       concat(preds k, l) pretend None), string(new()$Symbol))

suchThat(p:F, l:List(D -> Boolean)) ==
  retractIfCan(p)$Union(Symbol, "failed") case Symbol =>
  st(retract(p)@K, [f::Any for f in l])
  error "suchThat must be applied to symbols only"

——

— PMPREDFS.dotabb —

"PMPREDFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPREDFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMPREDFS" -> "FS"

——
package FSCINT FunctionSpaceComplexIntegration

--- FunctionSpaceComplexIntegration.input ---

)set break resume
)sys rm -f FunctionSpaceComplexIntegration.output
)spool FunctionSpaceComplexIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceComplexIntegration
--E 1

)spool
)lisp (bye)

---

--- FunctionSpaceComplexIntegration.help ---

====================================================================
FunctionSpaceComplexIntegration examples
====================================================================

Top-level complex function integration FunctionSpaceComplexIntegration provides functions for the indefinite integration of complex-valued functions.

See Also:
o )show FunctionSpaceComplexIntegration
CHAPTER 7. CHAPTER F

FunctionSpaceComplexIntegration (FSCINT)

Exports:
  complexIntegrate  internalIntegrate  internalIntegrate0

— package FSCINT FunctionSpaceComplexIntegration —

)abbrev package FSCINT FunctionSpaceComplexIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 11 June 1993
++ Description:
++ Top-level complex function integration
++ \spadtype{FunctionSpaceComplexIntegration} provides functions for the
++ indefinite integration of complex-valued functions.

FunctionSpaceComplexIntegration(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
    RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory,
    AlgebraicallyClosedFunctionSpace R)

SE ==> Symbol
G ==> Complex R
FG ==> Expression G
IR ==> IntegrationResult F

Exports ==> with
  internalIntegrate : (F, SE) -> IR
    ++ internalIntegrate(f, x) returns the integral of \spad{f(x)dx}
    ++ where x is viewed as a complex variable.
  internalIntegrate0: (F, SE) -> IR
    ++ internalIntegrate0 should be a local function, but is conditional.
  complexIntegrate : (F, SE) -> F
    ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
    ++ where x is viewed as a complex variable.
Implementation ==> add
import IntegrationTools(R, F)
import ElementaryIntegration(R, F)
import ElementaryIntegration(G, FG)
import AlgebraicManipulations(R, F)
import AlgebraicManipulations(G, FG)
import TrigonometricManipulations(R, F)
import IntegrationResultToFunction(R, F)
import IntegrationResultFunctions2(FG, F)
import ElementaryFunctionStructurePackage(R, F)
import ElementaryFunctionStructurePackage(G, FG)
import InnerTrigonometricManipulations(R, F, FG)

K2KG: Kernel F -> Kernel FG

K2KG k == retract(tan F2FG first argument k)@Kernel(FG)

complexIntegrate(f, x) ==
  removeConstantTerm(complexExpand internalIntegrate(f, x), x)

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
  and F has Join(LiouvilleFunctionCategory, RetractableTo SE) then
  import PatternMatchIntegration(R, F)
  internalIntegrate0(f, x) ==
    intPatternMatch(f, x, lfintegrate, pmComplexintegrate)
else internalIntegrate0(f, x) == lfintegrate(f, x)

internalIntegrate(f, x) ==
  f := distribute(f, x::F)
  any?(x1->has?(operator x1, "rtrig"),
    [k for k in tower(g := realElementary(f, x))
      | member?(x, variables(k::F))]
    )$List(Kernel F))$List(Kernel F) =>
    h := trigs2explogs(F2FG g, [K2KG k for k in tower f
      | is?(k, "tan"::SE) or is?(k, "cot"::SE)], [x])
    real?(g := FG2F h) =>
      internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)
    real?(g := FG2F(h := rootSimp(rischNormalize(h, x).func))) =>
      internalIntegrate0(g, x)
    map(FG2F, lfintegrate(h, x))
  internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)

— FSCINT.dotabb —

"FSCINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSCINT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
package FS2 FunctionSpaceFunctions2

--- FunctionSpaceFunctions2.input ---

)set break resume
)sys rm -f FunctionSpaceFunctions2.output
)spool FunctionSpaceFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceFunctions2
--E 1

)spool
)lisp (bye)

---

--- FunctionSpaceFunctions2.help ---

====================================================================
FunctionSpaceFunctions2 examples
====================================================================

Lifting of maps to function spaces. This package allows a mapping R \rightarrow S to be lifted to a mapping from a function space over R to a function space over S.

See Also:
o )show FunctionSpaceFunctions2
FunctionSpaceFunctions2 (FS2)

Exports:
map

— package FS2 FunctionSpaceFunctions2 —

)abbrev package FS2 FunctionSpaceFunctions2
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 3 May 1994
++ Description:
++ Lifting of maps to function spaces
++ This package allows a mapping R -> S to be lifted to a mapping
++ from a function space over R to a function space over S;

FunctionSpaceFunctions2(R, A, S, B): Exports == Implementation where
  R, S: Join(Ring, OrderedSet)
  A : FunctionSpace R
  B : FunctionSpace S

  K ==> Kernel A
  P ==> SparseMultivariatePolynomial(R, K)

Exports ==> with
  map: (R -> S, A) -> B
  "map(f, a) applies f to all the constants in R appearing in \spad{a}."

Implementation ==> add
  smpmap: (R -> S, P) -> B

smpmap(fn, p) ==
  map(x+->map(z+->map(fn, z)$ExpressionSpaceFunctions2(A,B),
         y+->fn(y)::B,p)_
  $PolynomialCategoryLifting(IndexedExponents K, K, R, P, B)

if R has IntegralDomain then
if $S$ has IntegralDomain then
    $\text{map}(f, x) == \text{smmap}(f, \text{numer } x) / \text{smmap}(f, \text{denom } x)$
else
    $\text{map}(f, x) == \text{smmap}(f, \text{numer } x) \ast (\text{recip}(\text{smmap}(f, \text{denom } x))::B)$
else
    $\text{map}(f, x) == \text{smmap}(f, \text{numer } x)$

package FSINT FunctionSpaceIntegration

--S 1 of 1
)show FunctionSpaceIntegration
--E 1

)spool
)lisp (bye)
Top-level real function integration FunctionSpaceIntegration provides functions for the indefinite integration of real-valued functions.

See Also:
o )show FunctionSpaceIntegration

---

FunctionSpaceIntegration (FSINT)

Exports:
 integrate

--- package FSINT FunctionSpaceIntegration ---

)abbrev package FSINT FunctionSpaceIntegration
 ++ Author: Manuel Bronstein
 ++ Date Created: 4 February 1988
 ++ Date Last Updated: 11 June 1993
 ++ Description:
 ++ Top-level real function integration
 ++ \spadtype{FunctionSpaceIntegration} provides functions for the
 ++ indefinite integration of real-valued functions.

FunctionSpaceIntegration(R, F):Exports == Implementation where
 R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
 RetractableTo Integer, LinearlyExplicitRingOver Integer)
 F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
 AlgebraicallyClosedFunctionSpace R)

B  ==> Boolean
G  ==> Complex R
K  ==> Kernel F
CHAPTER 7. CHAPTER F

\[ P \rightarrow \text{SparseMultivariatePolynomial}(R, K) \]
\[ \text{SE} \rightarrow \text{Symbol} \]
\[ \text{IR} \rightarrow \text{IntegrationResult} \]
\[ \text{FG} \rightarrow \text{Expression} \]
\[ \text{ALGOP} \rightarrow "%alg" \]
\[ \text{TANTEMP} \rightarrow "%temptan" :: \text{SE} \]

Exports \rightarrow \text{with}
\[ \text{integrate}: (\text{F}, \text{SE}) \rightarrow \text{Union}(\text{F}, \text{List}\text{F}) \]
\[ \rightarrow \text{integrates the integral of } \text{f(x)} dx \]
\[ \rightarrow \text{where x is viewed as a real variable.} \]

Implementation \rightarrow \text{add}
\[ \text{import IntegrationTools}(R, F) \]
\[ \text{import ElementaryIntegration}(R, F) \]
\[ \text{import ElementaryIntegration}(G, FG) \]
\[ \text{import AlgebraicManipulations}(R, F) \]
\[ \text{import TrigonometricManipulations}(R, F) \]
\[ \text{import IntegrationResultToFunction}(R, F) \]
\[ \text{import TranscendentalManipulations}(R, F) \]
\[ \text{import IntegrationResultFunctions2}(FG, F) \]
\[ \text{import FunctionSpaceComplexIntegration}(R, F) \]
\[ \text{import ElementaryFunctionStructurePackage}(R, F) \]
\[ \text{import InnerTrigonometricManipulations}(R, F, FG) \]
\[ \text{import PolynomialCategoryQuotientFunctions}(\text{IndexedExponents} K, K, R, \text{SparseMultivariatePolynomial}(R, K), F) \]

\[ K2KG : K \rightarrow \text{Kernel} \]
\[ \text{postSubst} : (\text{F}, \text{List F}, \text{List K}, \text{List K}, \text{SE}) \rightarrow \text{F} \]
\[ \text{rinteg} : (\text{IR}, \text{F}, \text{SE}, \text{B}, \text{List K}) \rightarrow \text{F} \]
\[ \text{mkPrimh} : (\text{F}, \text{SE}, \text{B}, \text{B}) \rightarrow \text{Union} (\text{F}, \text{List F}) \]
\[ \text{trans?} : \text{F} \rightarrow \text{B} \]
\[ \text{goComplex?} : (\text{B}, \text{List K}, \text{List K}) \rightarrow \text{B} \]
\[ \text{halfangle} : \text{F} \rightarrow \text{F} \]
\[ \text{Khalf} : \text{K} \rightarrow \text{F} \]
\[ \text{tan2temp} : \text{K} \rightarrow \text{K} \]

\[ \text{optemp} : \text{BasicOperator} \rightarrow \text{operator} (\text{TANTEMP}, 1) \]

\[ K2KG k = \text{retract} (\text{tan} \text{F2FG} \text{first argument} k) \text{Kernel}(\text{FG}) \]
\[ \text{tan2temp} k = \text{kernel} (\text{optemp}, \text{argument} k, \text{height} k) \]

\[ \text{trans?} f = \]
\[ \text{any?}(x1 \rightarrow \text{is?}(x1, "log" :: \text{SE}) \text{ or } \text{is?}(x1, "exp" :: \text{SE}) \text{ or } \text{is?}(x1, "atan" :: \text{SE}), \]
\[ \text{operators} f) \text{List} (\text{BasicOperator}) \]

\[ \text{mkPrimh}(f, x, h, \text{comp}) = \]
\[ f := \text{real} f \]
\[ \text{if} \text{comp} \text{then} f := \text{removeSinSq} f \]
\[ g := \text{mkPrim}(f, x) \]
h and trans? g => htrigs g

g
rinteg(i, f, x, h, comp) ==
not elem? i => integral(f, x)$F
empty? rest(1 := [mkPrimh(f, x, h, comp) for f in expand i]) => first 1
l

-- replace tan(a/2)**2 by (1-cos a)/(1+cos a) if tan(a/2) is in ltan
halfangle a ==
a := 2 * a
(1 - cos a) / (1 + cos a)

Khalf k ==
a := 2 * first argument k
sin(a) / (1 + cos a)

-- ltan = list of tangents in the integrand after real normalization
postSubst(f, lv, lk, comp, ltan, x) ==
for v in lv for k in lk repeat
  if ((u := retractIfCan(v)@Union(K, "failed")) case K) then
    if has?(operator(kk := u::K), ALGOP) then
      f := univariate(f, kk, minPoly kk) (kk::F)
      f := eval(f, [u::K], [k::F])
    if not(comp or empty? ltan) then
      ltemp := [tan2temp k for k in ltan]
      f := eval(f, ltan, [k::F for k in ltemp])
      f := eval(f, TANTEMP, 2, halfangle)
      f := eval(f, ltemp, [Khalf k for k in ltemp])
      removeConstantTerm(f, x)

-- can handle a single unnested tangent directly, otherwise go complex for now
-- l is the list of all the kernels containing x
-- ltan is the list of all the tangents in l
goComplex?(rt, l, ltan) ==
empty? ltan => rt
not empty? rest rest l

integrate(f, x) ==
not real? f => complexIntegrate(f, x)

f := distribute(f, x::F)
tf := [k for k in tower f | member?(x,variables(k::F))$List(SE)]$List(K)
ltf := select(x1+->is?(operator x1, "tan"::SE), tf)
ht := any?(x1+->has?(operator x1, "htrig"), tf)
rec := rischNormalize(realElementary(f, x), x)
g := rootSimp(rec.func)
tg := [k for k in tower g | member?(x, variables(k::F))]$List(K)
l tg := select(x1+->is?(operator x1, "tan"::SE), tg)
r tg := any?(x1+->has?(operator x1, "rtrig"), tg)
el := any?(x1+->has?(operator x1, "elem"), tg)
if (comp := goComplex?(rtg, tg, ltg)) then
  i := map(FG2F, lfintegrate(trigs2explogs(F2FG g,
    [K2KG k for k in tf | is?(k, "tan":SE) or
    is?(k, "cot":SE)], [x]), x))
else i := lfintegrate(g, x)
ltg := setDifference(ltg, ltf) -- tan's added by normalization
(u := rinteg(i, f, x, el and ht, comp)) case F =>
  postSubst(u::F, rec.vals, rec.kers, comp, ltg, x)
[postSubst(h, rec.vals, rec.kers, comp, ltg, x) for h in u::List(F)]
FunctionSpacePrimitiveElement examples
====================================================================
FunctionsSpacePrimitiveElement provides functions to compute
primitive elements in functions spaces;
See Also:
o )show FunctionSpacePrimitiveElement

---

FunctionSpacePrimitiveElement (FSPRMELT)

Exports:
primitiveElement

--- package FSPRMELT FunctionSpacePrimitiveElement ---

)abbrev package FSPRMELT FunctionSpacePrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Description:
++ FunctionsSpacePrimitiveElement provides functions to compute
++ primitive elements in functions spaces;

FunctionSpacePrimitiveElement(R, F):Exports == Implementation where
R: Join(IntegralDomain, OrderedSet, CharacteristicZero)
F: FunctionSpace R

SY => Symbol
P => Polynomial F
K => Kernel F
CHAPTER 7. CHAPTER F

UP ==> SparseUnivariatePolynomial F
REC ==> Record(primelt:F, poly:List UP, prim:UP)

Exports ==> with
primitiveElement: List F -> Record(primelt:F, poly:List UP, prim:UP)
  ++ primitiveElement([a1,...,an]) returns \spad{[a, [q1,...,qn], q]}
++ such that then \spad{k(a1,...,an) = k(a)},
++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
++ This operation uses the technique of
++ \spadglossSee{groebner bases}{Groebner basis}.
if F has AlgebraicallyClosedField then
primitiveElement: (F,F)->Record(primelt:F,pol1:UP,pol2:UP,prim:UP)
  ++ primitiveElement(a1, a2) returns \spad{[a, q1, q2, q]}
++ such that \spad{k(a1, a2) = k(a)},
++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
++ The minimal polynomial for a2 may involve a1, but the
++ minimal polynomial for a1 may not involve a2;
++ This operations uses \spadfun{resultant}.

Implementation ==> add
import PrimitiveElement(F)
import AlgebraicManipulations(R, F)
import PolynomialCategoryLifting(IndexedExponents K, K, R, SparseMultivariatePolynomial(R, K), P)

F2P: (F, List SY) -> P
K2P: (K, List SY) -> P

F2P(f, l) ==
  inv(denom(f)::F)*map((k1:K):P+->K2P(k1,l),(r1:R):P+->r1::F::P, numer f)
K2P(k, l) ==
  ((v := symbolIfCan k) case SY) and member?(v::SY, l) => v::SY::P
  k::F::P

primitiveElement l ==
  u := string(uu := new()$SY)
  vars := [concat(u, string i)::SY for i in 1..#l]
  vv := [kernel(v)$K :: F for v in vars]
  kers := [retract(a)@K for a in l]
  pols := [F2P(subst(ratDenom((minPoly k) v, kers), kers, vv), vars)
            for k in kers for v in vv]
  rec := primitiveElement(pols, vars, uu)
  [+/[c * a for c in rec.coef for a in l], rec.poly, rec.prim]
if F has AlgebraicallyClosedField then
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, SparseMultivariatePolynomial(R, K), F)

F2UP: (UP, K, UP) -> UP
getpoly: (UP, F) -> UP

F2UP(p, k, q) ==
  ans2:UP := 0
  while not zero? p repeat
    f := univariate(leadingCoefficient p, k)
    ans := ans + ((numer f) q) * monomial(inv(retract(denom f)@F), degree p)
    p := reductum p
  ans

primitiveElement(a1, a2) ==
  a := (aa := new()$SY)::F
  b := (bb := new()$SY)::F
  l := [aa, bb]$List(SY)
  p1 := minPoly(k1 := retract(a1)@K)
  p2 := map((z1:F):F+->subst(ratDenom(z1, [k1]), [k1], [a]),
             minPoly(retract(a2)@K))
  rec := primitiveElement(F2P(p1 a, l), aa, F2P(p2 b, l), bb)
  w := rec.coef1 * a1 + rec.coef2 * a2
  g := rootOf(rec.prim)
  zero?(rec.coef1) =>
    c2g := inv(rec.coef2 :: F) * g
    r := gcd(p1, univariate(p2 c2g, retract(a)@K, p1))
    q := getpoly(r, g)
    [w, q, rec.coef2 * monomial(1, 1)$UP, rec.prim]
  ic1 := inv(rec.coef1 :: F)
  gg := (ic1 * g)::UP - monomial(rec.coef2 * ic1, 1)$UP
  kg := retract(g)@K
  r := gcd(p1 gg, F2UP(p2, retract(a)@K, gg))
  q := getpoly(r, g)
  [w, monomial(ic1, 1)$UP - rec.coef2 * ic1 * q, q, rec.prim]

getpoly(r, g) ==
  -- one? degree r =>
  (degree r = 1) =>
    k := retract(g)@K
    univariate(-coefficient(r,0)/leadingCoefficient r,k,minPoly k)
  error "GCD not of degree 1"
package FSRED FunctionSpaceReduce

— FunctionSpaceReduce.input —

)set break resume
)sys rm -f FunctionSpaceReduce.output
)spool FunctionSpaceReduce.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceReduce
--E 1

)spool
)lisp (bye)

— FunctionSpaceReduce.help —

====================================================================
FunctionSpaceReduce examples
====================================================================

Reduction from a function space to the rational numbers
This package provides function which replaces transcendental kernels in a function space by random integers. The correspondence between the kernels and the integers is fixed between calls to new().

See Also:
o )show FunctionSpaceReduce
FunctionSpaceReduce (FSRED)

Exports:
  bringDown  newReduc

— package FSRED FunctionSpaceReduce —

)abbrev package FSRED FunctionSpaceReduce
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
  ++ Reduction from a function space to the rational numbers
  ++ This package provides function which replaces transcendental kernels
  ++ in a function space by random integers. The correspondence between
  ++ the kernels and the integers is fixed between calls to new().

FunctionSpaceReduce(R, F):Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F: FunctionSpace R

Z  ==> Integer
Q  ==> Fraction Integer
UP ==> SparseUnivariatePolynomial Q
K  ==> Kernel F
ALGOP => "%alg"

Exports == with
  bringDown: F -> Q
    ++ bringDown(f) \undocumented
  bringDown: (F, K) -> UP
    ++ bringDown(f,k) \undocumented
  newReduc : () -> Void
    ++ newReduc() \undocumented

Implementation ==> add
  import SparseUnivariatePolynomialFunctions2(F, Q)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, SparseMultivariatePolynomial(R, K), F)

K2Z : K -> F
redmap := table()$AssociationList(K, Z)
newReduc() ==
   for k in keys redmap repeat remove_!(k, redmap)
   void
bringDown(f, k) ==
   ff := univariate(f, k)
   (bc := extendedEuclidean(map(bringDown, denom ff),
      m := map(bringDown, minPoly k), 1)) case "failed" =>
      error "denominator is 0"
   (map(bringDown, numer ff) * bc.coef1) rem m
bringDown f ==
   retract(eval(f, lk := kernels f, [K2Z k for k in lk]))@Q
K2Z k ==
   has?(operator k, ALGOP) => error "Cannot reduce constant field"
   (u := search(k, redmap)) case "failed" =>
      setelt(redmap, k, random()$Z)::F
   u::Z::F

—— FSRED.dotabb ——
"FSRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSRED"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSRED" -> "FS"

package SUMFS FunctionSpaceSum

—— FunctionSpaceSum.input ——
)set break resume
)sys rm -f FunctionSpaceSum.output
)spool FunctionSpaceSum.output
FunctionSpaceSum (SUMFS)

Exports:
sum

---

<table>
<thead>
<tr>
<th>FunctionSpaceSum.help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computes sums of top-level expressions</td>
</tr>
<tr>
<td>See Also:</td>
</tr>
<tr>
<td>o )show FunctionSpaceSum</td>
</tr>
</tbody>
</table>

---

SUMFS

 FS ACF

Exports:
sum

---

---

| package SUMFS FunctionSpaceSum |
|--------------------------------|--
`CHAPTER 7. CHAPTER F

)abbrev package SUMFS FunctionSpaceSum
++ Author: Manuel Bronstein
++ Date Last Updated: 19 April 1991
++ Description:
++ Computes sums of top-level expressions

FunctionSpaceSum(R, F): Exports == Implementation where
R: Join(IntegralDomain, OrderedSet,
    RetractaibleTo Integer, LinearlyExplicitRingOver Integer)
F: Join(FunctionSpace R, CombinatorialOpsCategory,
    AlgebraicallyClosedField, TranscendentalFunctionCategory)

SE ==> Symbol
K ==> Kernel F

Exports ==> with
  sum: (F, SE) -> F
  ++ sum(a(n), n) returns A(n) such that A(n+1) - A(n) = a(n);
  sum: (F, SegmentBinding F) -> F
  ++ sum(f(n), n = a..b) returns f(a) + f(a+1) + ... + f(b);

Implementation ==> add
  import ElementaryFunctionStructurePackage(R, F)
  import GosperSummationMethod(IndexedExponents K, K, R,
    SparseMultivariatePolynomial(R, K), F)

  innersum: (F, K) -> Union(F, "failed")
  notRF? : (F, K) -> Boolean
  newk : () -> K

  newk() == kernel(new()$SE)

  sum(x:F, s:SegmentBinding F) ==
    k := kernel(variable s)$K
    (u := innersum(x, k)) case "failed" => summation(x, s)
    eval(u::F, k, 1 + hi segment s) - eval(u::F, k, lo segment s)

  sum(x:F, v:SE) ==
    (u := innersum(x, kernel(v)$K)) case "failed" => summation(x,v)
    u::F

  notRF?(f, k) ==
    for kk in tower f repeat
      member?(k, tower(kk::F)) and (symbolIfCan(kk) case "failed") =>
        return true
    false

  innersum(x, k) ==
    zero? x => 0
    notRF?(f := normalize(x / (x1 := eval(x, k, k::F - 1))), k) =>
"failed"
(u := GospersMethod(f, k, newk)) case "failed" => "failed"
x1 * eval(u::F, k, k::F - 1)

-- SUMFS.dotabb --

"SUMFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUMFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"SUMFS" -> "FS"
"SUMFS" -> "ACF"

package FS2EXPXP FunctionSpaceToExponentialExpansion

-- FunctionSpaceToExponentialExpansion.input --

)set break resume
)sys rm -f FunctionSpaceToExponentialExpansion.output
)spool FunctionSpaceToExponentialExpansion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceToExponentialExpansion
--E 1

)spool
)lisp (bye)

-- FunctionSpaceToExponentialExpansion.help --

====================================================================
FunctionSpaceToExponentialExpansion examples
====================================================================
This package converts expressions in some function space to exponential expansions.

See Also:
o show FunctionSpaceToExponentialExpansion

---

**FunctionSpaceToExponentialExpansion (FS2EXPXP)**

Exports:
localAbs exprToXXP

— package FS2EXPXP FunctionSpaceToExponentialExpansion —

)abbrev package FS2EXPXP FunctionSpaceToExponentialExpansion
++ Author: Clifton J. Williamson
++ Date Created: 17 August 1992
++ Date Last Updated: 2 December 1994
++ Description:
++ This package converts expressions in some function space to exponential
++ expansions.

FunctionSpaceToExponentialExpansion(R,FE,x,cen):_
     Exports == Implementation where
     R    : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
                LinearlyExplicitRingOver Integer)
     FE   : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
                FunctionSpace R)
     x    : Symbol
     cen  : FE
     B    ==> Boolean
     BOP  ==> BasicOperator
PACKAGE FS2EXPXP FUNKTIONSPACE TO EXPONENTIAL EXPANSION

Exports ==>

exprToXXP : (FE,B) -> XResult
++ exprToXXP(fcn, posCheck?) converts the expression \spad{fcn} to
++ an exponential expansion. If \spad{posCheck?} is true,
++ log's of negative numbers are not allowed nor are nth roots of
++ negative numbers with n even. If \spad{posCheck?} is false,
++ these are allowed.

localAbs: FE -> FE
++ localAbs(fcn) = \spad{abs(fcn)} or \spad{sqrt(fcn^2)} depending
++ on whether or not FE has a function \spad{abs}. This should be
++ a local function, but the compiler won't allow it.

Implementation ==> add

import FS2UPS -- conversion of functional expressions to Puiseux series
import EFUPXS -- partial transcendental functions on UPXS

ratIfCan : FE -> Union(RN,"failed")
stateSeriesProblem : (S,S) -> Result
stateProblem : (S,S) -> XResult
newElem : FE -> FE
smpElem : SMP -> FE
k2Elem : K -> FE
iExprToXXP : (FE,B) -> XResult
listToXXP : (L FE,B,XXP,(XXP,XXP) -> XXP) -> XResult
isNonTrivPower : FE -> Union(Record(val:FE,exponent:I),"failed")
negativePowerOK? : UPXS -> Boolean
powerToXXP : (FE,I,B) -> XResult
carefulNthRootIfCan : (UPXS,NNI,B) -> Result
nthRootXXPIfCan : (XXP,NNI,B) -> XResult
nthRootToXXP : (FE,NNI,B) -> XResult
genPowerToXXP : (L FE,B) -> XResult
kernelToXXP : (K,B) -> XResult
genExp : (UPXS,B) -> Result
exponential : (UPXS,B) -> XResult
expToXXP : (FE,B) -> XResult
genLog : (UPXS,B) -> Result
logToXXP : (FE,B) -> XResult
applyIfCan : (UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
applyBddIfCan : (FE,UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
tranToXXP : (K,FE,B) -> XResult
contOnReals? : S -> B
bddOnReals? : S -> B
opsInvolvingX : FE -> L BOP
opInOpList? : (SY,L BOP) -> B
exponential? : FE -> B
productOfNonZeroes? : FE -> B
atancotToXXP : (FE,FE,B,I) -> XResult

ZEROCOUNT : RN := 1000/1
-- number of zeroes to be removed when taking logs or nth roots

--% retractions
ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")

--% 'problems' with conversion
stateSeriesProblem(function,problem) ==
-- records the problem which occured in converting an expression
-- to a power series
[[function,problem]]

stateProblem(function,problem) ==
-- records the problem which occured in converting an expression
-- to an exponential expansion
[[function,problem]]

--% normalizations

newElem f ==
-- rewrites a functional expression; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp; all inverse
-- hyperbolic trig functions are expressed in terms of exp
-- and log
smpElem(numer f) / smpElem(denom f)

smpElem p == map(k2Elem,(x1:R):FE++>x1::FE,p)$PCL

k2Elem k ==
-- rewrites a kernel; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
null(args := [newElem a for a in argument k]) => k :: FE
iez := inv(ez := exp(z := first args))
sinz := sin z; cosz := cos z
is?(k,"tan" :: SY) => sinz / cosz
is?(k,"cot" :: SY) => cosz / sinz
is?(k,"sec" :: SY) => inv cosz
is?(k,"csc" :: SY) => inv sinz
is?(k,"sinh" :: SY) => (ez - iez) / (2 :: FE)
is?(k,"cosh" :: SY) => (ez + iez) / (2 :: FE)
is?(k,"tanh" :: SY) => (ez - iez) / (ez + iez)
is?(k,"coth" :: SY) => (ez + iez) / (ez - iez)
is?(k,"sech" :: SY) => 2 * inv(ez + iez)
is?(k,"csch" :: SY) => 2 * inv(ez - iez)
is?(k,"acosh" :: SY) => log(sqrt(z**2 - 1) + z)
is?(k,"atanh" :: SY) => log((z + 1) / (1 - z)) / (2 :: FE)
is?(k,"acoth" :: SY) => log((z + 1) / (z - 1)) / (2 :: FE)
is?(k,"asech" :: SY) => log((inv z) + sqrt(inv(z**2) - 1))
is?(k,"acsch" :: SY) => log((inv z) + sqrt(1 + inv(z**2)))
(operator k) args

--% general conversion function
exprToXXP(fcn,posCheck?) == iExprToXXP(newElem fcn,posCheck?)

iExprToXXP(fcn,posCheck?) ==
-- converts a functional expression to an exponential expansion
--!! The following line is commented out so that expressions of
--!! the form a**b will be normalized to exp(b * log(a)) even if
--!! 'a' and 'b' do not involve the limiting variable 'x'.
--!! - cjw 1 Dec 94
--not member?(x:variables fcn) => [monomial(fcn,0)$UPXS :: XXP]
(poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
[exprToUPS(fcn,false,"real:two sides")].%series :: XXP
(sum := isPlus fcn) case L(FE) =>
listToXXP(sum::L(FE),posCheck?,0,(y1:XXP,y2:XXP):XXP +-> y1+y2)
(prod := isTimes fcn) case L(FE) =>
listToXXP(prod :: L(FE),posCheck?,1,(y1:XXP,y2:XXP):XXP +-> y1*y2)
(expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
power := expt :: Record(val:FE,exponent:I)
powerToXXP(power.val,power.exponent,posCheck?)
(ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
kernelToXXP(ker :: K,posCheck?)
error "exprToXXP: neither a sum, product, power, nor kernel"

--% sums and products

listToXXP(list,posCheck?,ans,op) ==
  -- converts each element of a list of expressions to an exponential
  -- expansion and returns the sum of these expansions, when 'op' is +
  -- and 'ans' is 0, or the product of these expansions, when 'op' is *
  -- and 'ans' is 1
while not null list repeat
  (term := iExprToXXP(first list,posCheck?)) case %problem =>
    return term
  ans := op(ans,term.%expansion)
  list := rest list
[ans]

--% nth roots and integral powers

isNonTrivPower fcn ==
  -- is the function a power with exponent other than 0 or 1?
  (expt := isPower fcn) case "failed" => "failed"
  power := expt :: Record(val:FE,exponent:I)
  -- one? power.exponent => "failed"
  (power.exponent = 1) => "failed"
  power

negativePowerOK? upxs ==
  -- checks the lower order coefficient of a Puiseux series;
  -- the coefficient may be inverted only if
  -- (a) the only function involving x is 'log', or
  -- (b) the lowest order coefficient is a product of exponentials
  -- and functions not involving x
  deg := degree upxs
  if (coef := coefficient(upxs,deg)) = 0 then
    deg := order(upxs,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(upxs,deg)) = 0 =>
      error "inverse of series with many leading zero coefficients"
    xOpList := opsInvolvingX coef
    -- only function involving x is 'log'
    (null xOpList) => true
    (null rest xOpList and is?(first xOpList,"log" :: SY)) => true
    -- lowest order coefficient is a product of exponentials and
    -- functions not involving x
    productOfNonZeroes? coef => true
    false

powerToXXP(fcn,n,posCheck?) ==
  -- converts an integral power to an exponential expansion
\( (b := \text{iExprToXXP}(fcn,\text{posCheck?})) \) case %\text{problem} => b

\( \text{xxp := b.}\text{%expansion} \)

\( n > 0 \Rightarrow [\text{xxp} ** n] \)

-- a Puiseux series will be reciprocated only if \( n < 0 \) and
-- numerator of \( \text{xxp} \) has exactly one monomial

\( \text{negativePowerOK? leadingCoefficient num} \Rightarrow \)

\( \text{(rec := recip num) case "failed" => error "FS2EXPXP: can’t happen"} \)

\( \text{nn := (-n) :: NNI} \)

\( [(((\text{denom xxp}) ** nn) * ((\text{rec} :: \text{UPXSSING}) ** nn)) :: XXP] \)

-- !! we may want to create a fraction instead of trying to
-- !! reciprocate the numerator

\( \text{stateProblem("inv","lowest order coefficient involves x")} \)

\( \text{carefulNthRootIfCan(ups,n,\text{posCheck?})} = \)

-- similar to \( \text{nthRootIfCan} \), but it is fussy about the series
-- it takes as an argument. If \( n \) is EVEN and \( \text{posCheck?} \)
-- is true then the leading coefficient of the series must
-- be POSITIVE. In this case, if \( \text{rightOnly?} \) is false, the
-- order of the series must be zero. The idea is that the
-- series represents a real function of a real variable, and
-- we want a unique real nth root defined on a neighborhood
-- of zero.

\( n < 1 \Rightarrow \text{error "nthRoot: n must be positive"} \)

\( \text{deg := degree ups} \)

\( \text{if (coef := coefficient(ups,deg)) = 0 then} \)

\( \text{deg := order(ups,deg + ZEROCOUNT :: Expon)} \)

\( \text{(coef := coefficient(ups,deg)) = 0 =>} \)

\( \text{error "log of series with many leading zero coefficients"} \)

-- if \( \text{posCheck?} \) is true, we do not allow nth roots of negative
-- numbers when \( n \) in even

\( \text{if even?(n :: I) then} \)

\( \text{if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then} \)

\( \text{(signum :: I) = -1 =>} \)

\( \text{return stateSeriesProblem("nth root","root of negative number")} \)

\( \text{(ans := nthRootIfCan(ups,n)) case "failed" =>} \)

\( \text{stateSeriesProblem("nth root","no nth root")} \)

\( \text{[ans :: UPXS]} \)

\( \text{nthRootXXPIfCan(xxp,n,\text{posCheck?})} = \)

\( \text{num := numer xxp; den := denom xxp} \)

\( \text{not zero?(reductum num) or not zero?(reductum den) =>} \)

\( \text{stateProblem("nth root","several monomials in numerator or denominator")} \)

\( \text{nInv : RN := 1/n} \)

\( \text{newNum :=} \)

\( \text{coef : UPXS :=} \)

\( \text{root := carefulNthRootIfCan(leadingCoefficient num,n,\text{posCheck?})} \)

\( \text{root case %\text{problem} => return [root.}\%\text{problem}] \)

\( \text{root.}\%\text{series} \)

\( \text{deg := (nInv :: FE) * (degree num)} \)
monomial(coef,deg)

newDen :=
coef : UPXS :=
root := carefulNthRootIfCan(leadingCoefficient den,n,posCheck?)
root case %problem => return [root.%problem]
root.%series
deg := (nInv :: FE) * (degree den)
monomial(coef,deg)
[newNum/newDen]

nthRootToXXP(arg,n,posCheck?) ==
-- converts an nth root to a power series
-- this is not used in the limit package, so the series may
-- have non-zero order, in which case nth roots may not be unique
(result := iExprToXXP(arg,posCheck?)) case %problem => [result.%problem]
ans := nthRootXXPIfCan(result.%expansion,n,posCheck?)
ans case %problem => [ans.%problem]
[ans.%expansion]

--% general powers f(x) ** g(x)

genPowerToXXP(args,posCheck?) ==
-- converts a power f(x) ** g(x) to an exponential expansion
(logBase := logToXXP(first args,posCheck?)) case %problem =>
logBase
(expon := iExprToXXP(second args,posCheck?)) case %problem =>
expon
xxp := (expon.%expansion) * (logBase.%expansion)
(f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
stateProblem("exp","multiply nested exponential")
exponential(f,posCheck?)

--% kernels

kernelToXXP(ker,posCheck?) ==
-- converts a kernel to a power series
(sym := symbolIfCan(ker)) case Symbol =>
(sym :: Symbol) = x => [monomial(1,1)$UPXS :: XXP]
[monomial(ker :: FE,0)$UPXS :: XXP]
empty?(args := argument ker) => [monomial(ker :: FE,0)$UPXS :: XXP]
empty? rest args =>
arg := first args
is?(ker,"%paren" :: Symbol) => iExprToXXP(arg,posCheck?)
is?(ker,"log" :: Symbol) => logToXXP(arg,posCheck?)
is?(ker,"exp" :: Symbol) => expToXXP(arg,posCheck?)
tranToXXP(ker,arg,posCheck?)
is?(ker,"%power" :: Symbol) => genPowerToXXP(args,posCheck?)
is?(ker,"nthRoot" :: Symbol) =>
n := retract(second args)@I
nthRootToXXP(first args,n :: NNI,posCheck?)
stateProblem(string name ker,"unknown kernel")

-- % exponentials and logarithms

genExp(ups,posCheck?) ==
  -- If the series has order zero and the constant term a0 of the
  -- series involves x, the function tries to expand exp(a0) as
  -- a power series.
  (deg := order(ups,1)) < 0 =>
    -- this "can't happen"
    error "exp of function with singularity"
  deg > 0 => [exp(ups)]
  lc := coefficient(ups,0); varOpList := opsInvolvingX lc
  not opInOpList?("log" :: Symbol,varOpList) => [exp(ups)]
  -- try to fix exp(lc) if necessary
  expCoef := normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
  result := exprToGenUPS(expCoef,posCheck?,"real:right side")$FS2UPS
  --!! will deal with problems in limitPlus in EXPEXPAN
  --result case %problem => result
  result case %problem => [exp(ups)]
  [(result.%series) * exp(ups - monomial(lc,0))]
CHAPTER 7.

```
((rat := ratIfCan coef) case RN) =>
  (rat :: RN) < 0 => true
false
false
logTerm : FE :=
  mon : FE := (x :: FE) - (cen :: FE)
pow : FE := mon ** (deg :: FE)
negRat? => log(coef * pow)
term1 : FE := (deg :: FE) * log(mon)
log(coef) + term1

[monomial(logTerm,0)$UPXS + log(ups/lt)]

logToXXP(arg,posCheck?) ==
  (result := iExprToXXP(arg,posCheck?)) case %problem => result
xxp := result.%expansion
num := numer xxp; den := denom xxp
not zero?(reductum num) or not zero?(reductum den) =>
  stateProblem("log","several monomials in numerator or denominator")
numCoefLog : UPXS :=
  (res := genLog(leadingCoefficient num,posCheck?)) case %problem =>
    return [res.%problem]
res.%series
denCoefLog : UPXS :=
  (res := genLog(leadingCoefficient den,posCheck?)) case %problem =>
    return [res.%problem]
res.%series
numLog := (exponent degree num) + numCoefLog
denLog := (exponent degree den) + denCoefLog --?? num?
[(numLog - denLog) :: XXP]

--% other transcendental functions

applyIfCan(fcn, arg, fcnName, posCheck?) ==
  -- converts fcn(arg) to an exponential expansion
  (xxpArg := iExprToXXP(arg,posCheck?)) case %problem => xxpArg
xxp := xxpArg.%expansion
(f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
  stateProblem(fcnName,"multiply nested exponential")
upxs := f :: UPXS
(deg := order(upxs,1)) < 0 =>
  stateProblem(fcnName,"essential singularity")
deg > 0 => [fcn(upxs) :: UPXS :: XXP]
1c := coefficient(upxs,0); xOpList := opsInvolvingX 1c
null xOpList => [fcn(upxs) :: UPXS :: XXP]
opInOpList?("log" :: SY,xOpList) =>
  stateProblem(fcnName,"logs in constant coefficient")
contOnReals? fcnName => [fcn(upxs) :: UPXS :: XXP]
stateProblem(fcnName,"x in constant coefficient")

applyBddIfCan(fe, fcn, arg, fcnName, posCheck?) ==
```
-- converts fcn(arg) to a generalized power series, where the
-- function fcn is bounded for real values
-- if fcn(arg) has an essential singularity as a complex
-- function, we return fcn(arg) as a monomial of degree 0
(xxpArg := iExprToXXP(arg, posCheck?) case %problem =>
  trouble := xxpArg.%problem
  trouble.prob = "essential singularity" => [monomial(fe,0)$UPXS :: XXP]
  xxpArg
xxp := xxpArg.%expansion
(f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
  stateProblem("exp","multiply nested exponential")
(ans := fcn(f :: UPXS)) case "failed" => [monomial(fe,0)$UPXS :: XXP]
[ans :: UPXS :: XXP]

CONTFCNS : L S := ["sin","cos","atan","acot","exp","asinh"]
-- functions which are defined and continuous at all real numbers
BDDFCNS : L S := ["sin","cos","atan","acot"]
-- functions which are bounded on the reals
contOnReals? fcn == member?(fcn,CONTFCNS)
bddOnReals? fcn == member?(fcn,BDDFCNS)

opsInvolvingX fcn ==
  opList := [op for k in tower fcn | unary?(op := operator k) _
             and member?(x,variables first argument k)]
  removeDuplicates opList

opInOpList?(name,opList) ==
  for op in opList repeat
    is?(op,name) => return true
  false

exponential? fcn ==
-- is 'fcn' of the form exp(f)?
  (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
    is?(ker :: K,"exp" :: Symbol)
  false

productOfNonZeroes? fcn ==
-- is 'fcn' a product of non-zero terms, where 'non-zero'
-- means an exponential or a function not involving x
exponential? fcn => true
  (prod := isTimes fcn) case "failed" => false
  for term in (prod :: L(FE)) repeat
    (not exponential? term) and member?(x,variables term) =>
      return false
  true

tranToXXP(ker, arg, posCheck?) ==
-- converts op(arg) to a power series for certain functions
-- op in trig or hyperbolic trig categories
-- N.B. when this function is called, 'k2elem' will have been
-- applied, so the following functions cannot appear:
-- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
-- acosh, atanh, acoth, asech, acsch
is?(ker,"sin" :: SY) =>$ applyBddIfCan(ker :: FE,sinIfCan,arg,"sin",posCheck?)
is?(ker,"cos" :: SY) =>$ applyBddIfCan(ker :: FE,cosIfCan,arg,"cos",posCheck?)
is?(ker,"asin" :: SY) =>$ applyIfCan(asinIfCan,arg,"asin",posCheck?)
is?(ker,"acos" :: SY) =>$ applyIfCan(acosIfCan,arg,"acos",posCheck?)
is?(ker,"atan" :: SY) =>$ atancotToXXP(ker :: FE,arg,posCheck?,1)
is?(ker,"acot" :: SY) =>$ atancotToXXP(ker :: FE,arg,posCheck?,-1)
is?(ker,"asec" :: SY) =>$ applyIfCan(asecIfCan,arg,"asec",posCheck?)
is?(ker,"acsc" :: SY) =>$ applyIfCan(acscIfCan,arg,"acsc",posCheck?)
is?(ker,"asinh" :: SY) =>$ applyIfCan(asinhIfCan,arg,"asinh",posCheck?)
stateProblem(string name ker,"unknown kernel")

if FE has abs: FE -> FE then
  localAbs fcn == abs fcn
else
  localAbs fcn == sqrt(fcn * fcn)

signOfExpression: FE -> FE
signOfExpression arg == localAbs(arg)/arg

atancotToXXP(fe,arg,posCheck?,plusMinus) =>$  -- converts atan(f(x)) to a generalized power series
  atanFlag : String := "real: right side"; posCheck? : Boolean := true
  (result : exprToGenUPS(arg,posCheck?,atanFlag)$FS2UPS) case %problem =>$  trouble := result.%problem
trouble.prob = "essential singularity" => [monomial(fe,0)$UPXS :: XXP]
  [result.%problem]
ups := result.%series; coef := coefficient(ups,0)
  -- series involves complex numbers
  (ord := order(ups,0)) = 0 and coef * coef = -1 =>$  y := differentiate(ups)/(1 + ups*ups)
yCoef := coefficient(y,-1)
  [(monomial(log yCoef,0)+integrate(y - monomial(yCoef,-1)$UPXS)) :: XXP]
cc : FE :=
  ord < 0 =>$  (rn := ratIfCan(ord :: FE)) case "failed" =>
-- this condition usually won't occur because exponents will
-- be integers or rational numbers
return stateProblem("atan","branch problem")
lc := coefficient(ups,ord)
(signum := sign(lc)$SIGNEF) case "failed" =>
  -- can't determine sign
  posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
  plusMinus = 1 => posNegPi2
  pi()/(2 :: FE) - posNegPi2
(n := signum :: Integer) = -1 =>
  plusMinus = 1 => -pi()/(2 :: FE)
  pi()
  plusMinus = 1 => pi()/(2 :: FE)
0
atan coef
[((cc :: UPXS) + integrate(differentiate(ups)/(1 + ups*ups))) :: XXP]
This package converts expressions in some function space to power series in a variable \( x \) with coefficients in that function space. The function \( \text{exprToUPS} \) converts expressions to power series whose coefficients do not contain the variable \( x \). The function \( \text{exprToGenUPS} \) converts functional expressions to power series whose coefficients may involve functions of \( \log(x) \).

See Also:
- \( \)show FunctionSpaceToUnivariatePowerSeries
++ series in a variable \( x \) with coefficients in that function space.
++ The function `\spadfun{exprToUPS}` converts expressions to power series
++ whose coefficients do not contain the variable \( x \). The function
++ `\spadfun{exprToGenUPS}` converts functional expressions to power series
++ whose coefficients may involve functions of \( \spad{\log(x)} \).

FunctionSpaceToUnivariatePowerSeries(R,FE,Expon,UPS,TRAN,x):_
Exports == Implementation where
   R : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
         LinearlyExplicitRingOver Integer)
   FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
         FunctionSpace R)
      with
         coerce: Expon -> %
         ++ coerce(e) converts an 'exponent' \( e \) to an 'expression'
   Expon : OrderedRing
   UPS : Join(UnivariatePowerSeriesCategory(FE,Expon),Field,_
         TranscendentalFunctionCategory)
      with
         differentiate: % -> %
         ++ differentiate(x) returns the derivative of \( x \) since we
         ++ need to be able to differentiate a power series
         integrate: % -> %
         ++ integrate(x) returns the integral of \( x \) since
         ++ we need to be able to integrate a power series
   TRAN : PartialTranscendentalFunctions UPS
   x : Symbol
   B ==> Boolean
   BOP ==> BasicOperator
   I ==> Integer
   NNI ==> NonNegativeInteger
   K ==> Kernel FE
   L ==> List
   RN ==> Fraction Integer
   S ==> String
   SY ==> Symbol
   PCL ==> PolynomialCategoryLifting(IndexedExponents K,K,R,SMP,FE)
   POL ==> Polynomial R
   SMP ==> SparseMultivariatePolynomial(R,K)
   SUP ==> SparseUnivariatePolynomial Polynomial R
   Problem ==> Record(func:String,prob:String)
   Result ==> Union(%series:UPS,%problem:Problem)
   SIGNEF ==> ElementaryFunctionSign(R,FE)

Exports == with
   exprToUPS : (FE,B,S) -> Result
      ++ exprToUPS(fcn,posCheck?,atanFlag) converts the expression
      ++ \spad{fcn} to a power series. If \spad{posCheck?} is true,
      ++ log's of negative numbers are not allowed nor are nth roots of
      ++ negative numbers with n even. If \spad{posCheck?} is false,
CHAPTER 7. CHAPTER F

++ these are allowed. \spad{atanFlag} determines how the case
++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
++ The possible values of \spad{atanFlag} are \spad{"complex"},
++ \spad{"real: two sides"}, \spad{"real: left side"},
++ \spad{"real: right side"}, and \spad{"just do it"}.
++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
++ will be computed because, viewed as a function of a complex
++ variable, \spad{atan(f(x))} has an essential singularity.
++ Otherwise, the sign of the leading coefficient of the series
++ expansion of \spad{f(x)} determines the constant coefficient
++ in the series expansion of \spad{atan(f(x))}. If this sign cannot
++ be determined, a series expansion is computed only when
++ \spad{atanFlag} is \spad{"just do it"}. When the leading term
++ in the series expansion of \spad{f(x)} is of odd degree (or is a
++ rational degree with odd numerator), then the constant coefficient
++ in the series expansion of \spad{atan(f(x))} for values to the
++ left differs from that for values to the right. If \spad{atanFlag}
++ is \spad{"real: two sides"}, no series expansion will be computed.
++ If \spad{atanFlag} is \spad{"real: left side"}, the constant
++ coefficient for values to the left will be used and if \spad{atanFlag}
++ \spad{"real: right side"}, the constant coefficient for values to the
++ right will be used.
++ If there is a problem in converting the function to a power series,
++ a record containing the name of the function that caused the problem
++ and a brief description of the problem is returned.
++ When expanding the expression into a series it is assumed that
++ the series is centered at 0. For a series centered at a, the
++ user should perform the substitution \spad{x -> x + a} before calling
++ this function.

exprToGenUPS : (FE,B,S) -> Result
++ exprToGenUPS(fcn,posCheck?,atanFlag) converts the expression
++ \spad{fcn} to a generalized power series. If \spad{posCheck?}
++ is true, log's of negative numbers are not allowed nor are nth roots
++ of negative numbers with n even. If \spad{posCheck?} is false,
++ these are allowed. \spad{atanFlag} determines how the case
++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
++ The possible values of \spad{atanFlag} are \spad{"complex"},
++ \spad{"real: two sides"}, \spad{"real: left side"},
++ \spad{"real: right side"}, and \spad{"just do it"}.
++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
++ will be computed because, viewed as a function of a complex
++ variable, \spad{atan(f(x))} has an essential singularity.
++ Otherwise, the sign of the leading coefficient of the series
++ expansion of \spad{f(x)} determines the constant coefficient
++ in the series expansion of \spad{atan(f(x))}. If this sign cannot
++ be determined, a series expansion is computed only when
++ \spad{atanFlag} is \spad{"just do it"}. When the leading term
++ in the series expansion of \spad{f(x)} is of odd degree (or is a
++ rational degree with odd numerator), then the constant coefficient
++ in the series expansion of \spad{atan(f(x))} for values to the
++ left differs from that for values to the right. If \spad{atanFlag}
++ is \spad{"real: two sides"}, no series expansion will be computed.
++ If \spad{atanFlag} is \spad{"real: left side"} the constant
++ coefficient for values to the left will be used and if \spad{atanFlag}
++ \spad{"real: right side"} the constant coefficient for values to the
++ right will be used.
++ If there is a problem in converting the function to a power
++ series, we return a record containing the name of the function
++ that caused the problem and a brief description of the problem.
++ When expanding the expression into a series it is assumed that
++ the series is centered at 0. For a series centered at a, the
++ user should perform the substitution \spad{x -> x + a} before calling
++ this function.

\textbf{localAbs: \texttt{FE} \rightarrow \texttt{FE}}
++ \texttt{localAbs(fcn) = abs(fcn)} or \texttt{sqrt(fcn**2)} depending
++ on whether or not \texttt{FE} has a function \texttt{abs}. This should be
++ a local function, but the compiler won’t allow it.

\textbf{Implementation ==> add}

\begin{verbatim}
ratIfCan : \texttt{FE} \rightarrow \texttt{Union(RN,\texttt{\"failed\")

carefulNthRootIfCan : \texttt{(UPS,NNI,B,B)} \rightarrow \texttt{Result

stateProblem : \texttt{(S,S)} \rightarrow \texttt{Result

polyToUPS : \texttt{SUP} \rightarrow \texttt{UPS

listToUPS : \texttt{(L FE,(FE,B,S) \rightarrow Result,B,S,UPS,(UPS,UPS) \rightarrow UPS)}

\rightarrow \texttt{Result

isNonTrivPower : \texttt{FE} \rightarrow \texttt{Union(Record(val:FE,exponent:I),\texttt{\"failed\")}

powerToUPS : \texttt{(FE,I,B,S)} \rightarrow \texttt{Result

kernelToUPS : \texttt{(K,B,S)} \rightarrow \texttt{Result

nthRootToUPS : \texttt{(FE,NNI,B,S)} \rightarrow \texttt{Result

logToUPS : \texttt{(FE,B,S)} \rightarrow \texttt{Result

atancotToUPS : \texttt{(FE,B,S,I)} \rightarrow \texttt{Result

applyIfCan : \texttt{(UPS \rightarrow Union(UPS,\texttt{\"failed\}),FE,S,B,S)} \rightarrow \texttt{Result

tranToUPS : \texttt{(K,FE,B,S)} \rightarrow \texttt{Result

powToUPS : \texttt{(L FE,B,S)} \rightarrow \texttt{Result

newElem : \texttt{FE \rightarrow FE

smpElem : \texttt{SMP \rightarrow FE

k2Elem : \texttt{K \rightarrow FE

contUnReals? : \texttt{S \rightarrow B

bdddUnReals? : \texttt{S \rightarrow B

iExprToGenUPS : \texttt{(FE,B,S)} \rightarrow \texttt{Result

opsInvolvingX : \texttt{FE \rightarrow L BOP

opInOpList? : \texttt{(SY,L BOP)} \rightarrow \texttt{B

exponential? : \texttt{FE \rightarrow B

productOfNonZeroes? : \texttt{FE \rightarrow B

powerToGenUPS : \texttt{(FE,I,B,S)} \rightarrow \texttt{Result

kernelToGenUPS : \texttt{(K,B,S)} \rightarrow \texttt{Result

nthRootToGenUPS : \texttt{(FE,NNI,B,S)} \rightarrow \texttt{Result

logToGenUPS : \texttt{(FE,B,S)} \rightarrow \texttt{Result
\end{verbatim}
expToGenUPS : (FE,B,S) -> Result
expGenUPS : (UPS,B,S) -> Result
atanToGenUPS : (FE,FE,B,S,I) -> Result
genUPSApplyIfCan : (UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
applyBddIfCan : (FE,UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
tranToGenUPS : (K,FE,B,S) -> Result
powToGenUPS : (L FE,B,S) -> Result

ZEROCOUNT : I := 1000
-- number of zeroes to be removed when taking logs or nth roots

ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")
carefulNthRootIfCan(ups,n,posCheck?,rightOnly?) ==
  -- similar to 'nthRootIfCan', but it is fussy about the series
  -- it takes as an argument. If 'n' is EVEN and 'posCheck'?
  -- is truem then the leading coefficient of the series must
  -- be POSITIVE. In this case, if 'rightOnly?' is false, the
  -- order of the series must be zero. The idea is that the
  -- series represents a real function of a real variable, and
  -- we want a unique real nth root defined on a neighborhood
  -- of zero.
  n < 1 => error "nthRoot: n must be positive"
  deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    deg := order(ups,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(ups,deg)) = 0 =>
      error "log of series with many leading zero coefficients"
  -- if 'posCheck?' is true, we do not allow nth roots of negative
  -- numbers when n in even
  if even?(n :: I) then
    if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then
      (signum :: I) = -1 =>
        return stateProblem("nth root","negative leading coefficient")
    not rightOnly? and not zero? deg => -- nth root not unique
      return stateProblem("nth root","series of non-zero order")
    (ans := nthRootIfCan(ups,n)) case "failed" =>
      stateProblem("nth root","no nth root")
    [ans :: UPS]
  stateProblem(function,problem) ==
  -- records the problem which occured in converting an expression
  -- to a power series
  [[function,problem]]

exprToUPS(fcn,posCheck?,atanFlag) ==
  -- converts a functional expression to a power series
  ---!! The following line is commented out so that expressions of
  ---!! the form a**b will be normalized to exp(b * log(a)) even if
  ---!! 'a' and 'b' do not involve the limiting variable 'x'.
--!! - cjw 1 Dec 94
--not member?(x,variables fcn) => [monomial(fcn,0)]
(poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
[polyToUPS univariate(poly :: POL,x)]
(sum := isPlus fcn) case L(FE) =>
listToUPS(sum :: L(FE),exprToUPS,posCheck?,atanFlag,0,
(y1,y2) +-> y1 + y2)
(prod := isTimes fcn) case L(FE) =>
listToUPS(prod :: L(FE),exprToUPS,posCheck?,atanFlag,1,
(y1,y2) +-> y1 * y2)
(expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
power := expt :: Record(val:FE,exponent:I)
powerToUPS(power.val,power.exponent,posCheck?,atanFlag)
(ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
kernelToUPS(ker :: K,posCheck?,atanFlag)
error "exprToUPS: neither a sum, product, power, nor kernel"

polyToUPS poly ==
-- converts a polynomial to a power series
zero? poly => 0
-- we don't start with 'ans := 0' as this may lead to an
-- enormous number of leading zeroes in the power series
deg := degree poly
coeff := leadingCoefficient(poly) :: FE
ans := monomial(coeff,deg :: Expon)$UPS
poly := reductum poly
while not zero? poly repeat
  deg := degree poly
coeff := leadingCoefficient(poly) :: FE
  ans := ans + monomial(coeff,deg :: Expon)$UPS
  poly := reductum poly
ans

listToUPS(list,feToUPS,posCheck?,atanFlag,ans,op) ==
-- converts each element of a list of expressions to a power
-- series and returns the sum of these series, when 'op' is +
-- and 'ans' is 0, or the product of these series, when 'op' is *
-- and 'ans' is 1
while not null list repeat
  (term := feToUPS(first list,posCheck?,atanFlag)) case %problem =>
    return term
  ans := op(ans,term.%series)
  list := rest list
[ans]

isNonTrivPower fcn ==
-- is the function a power with exponent other than 0 or 1?
(expt := isPower fcn) case "failed" => "failed"
power := expt :: Record(val:FE,exponent:I)
-- one? power.exponent => "failed"
(power.exponent = 1) => "failed"

powerToUPS(fcn,n,posCheck?,atanFlag) ==
  -- converts an integral power to a power series
  (b := exprToUPS(fcn,posCheck?,atanFlag)) case %problem => b
  n > 0 => [(b.%series) ** n]
  -- check lowest order coefficient when n < 0
  ups := b.%series; deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    deg := order(ups,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(ups,deg)) = 0 =>
      error "inverse of series with many leading zero coefficients"
  [ups ** n]

kernelToUPS(ker,posCheck?,atanFlag) ==
  -- converts a kernel to a power series
  (sym := symbolIfCan(ker)) case Symbol =>
    (sym :: Symbol) = x => [monomial(1,1)]
    [monomial(ker :: FE,0)]
  empty?(args := argument ker) => [monomial(ker :: FE,0)]
  not member?(x, variables(ker :: FE)) => [monomial(ker :: FE,0)]
  empty? rest args =>
    arg := first args
    is?(ker,"abs" :: Symbol) =>
      nthRootToUPS(arg*arg,2,posCheck?,atanFlag)
    is?(ker,"%paren" :: Symbol) => exprToUPS(arg,posCheck?,atanFlag)
    is?(ker,"log" :: Symbol) => logToUPS(arg,posCheck?,atanFlag)
    is?(ker,"exp" :: Symbol) =>
      applyIfCan(expIfCan,arg,"exp",posCheck?,atanFlag)
    tranToUPS(ker,arg,posCheck?,atanFlag)
    is?(ker,"%power" :: Symbol) => powToUPS(args,posCheck?,atanFlag)
    is?(ker,"nthRoot" :: Symbol) =>
      n := retract(second args)@I
      nthRootToUPS(first args,n :: NNI,posCheck?,atanFlag)
    stateProblem(string name ker,"unknown kernel")

nthRootToUPS(arg,n,posCheck?,atanFlag) ==
  -- converts an nth root to a power series
  -- this is not used in the limit package, so the series may
  -- have non-zero order, in which case nth roots may not be unique
  (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
  ans := carefulNthRootIfCan(result.%series,n,posCheck?,false)
  ans case %problem => ans
  [ans.%series]

logToUPS(arg,posCheck?,atanFlag) ==
  -- converts a logarithm log(f(x)) to a power series
  -- f(x) must have order 0 and if 'posCheck?' is true,
  -- then f(x) must have a non-negative leading coefficient
(result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
ups := result.%series
not zero? order(ups,1) =>
  stateProblem("log","series of non-zero order")
coef := coefficient(ups,0)
-- if 'posCheck?' is true, we do not allow logs of negative numbers
if posCheck? then
  if ((signum := sign(coef)$SIGNEF) case I) then
    (signum :: I) = -1 =>
      return stateProblem("log","negative leading coefficient")

if FE has abs: FE -> FE then
  localAbs fcn == abs fcn
else
  localAbs fcn == sqrt(fcn * fcn)

signOfExpression: FE -> FE
signOfExpression arg == localAbs(arg)/arg

atancotToUPS(arg,posCheck?,atanFlag,plusMinus) ==
  -- converts atan(f(x)) to a power series
  (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
  ups := result.%series; coef := coefficient(ups,0)
  (ord := order(ups,0)) = 0 and coef * coef = -1 =>
    -- series involves complex numbers
    return stateProblem("atan","logarithmic singularity")
  cc : FE :=
    ord < 0 =>
      atanFlag = "complex" =>
        return stateProblem("atan","essential singularity")
      (rn := ratIfCan(ord :: FE)) case "failed" =>
        -- this condition usually won't occur because exponents will
        -- be integers or rational numbers
        return stateProblem("atan","branch problem")
    if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
      -- expansions to the left and right of zero have different
      -- constant coefficients
      return stateProblem("atan","branch problem")
    lc := coefficient(ups,ord)
    (signum := sign(lc)$SIGNEF) case "failed" =>
      -- can't determine sign
      atanFlag = "just do it" =>
        plusMinus = 1 => pi()/(2 :: FE)
        0
        posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
        plusMinus = 1 => posNegPi2
        pi()/(2 :: FE) - posNegPi2
        --return stateProblem("atan","branch problem")
    left? : B := atanFlag = "real: left side"; n := signum :: Integer
(left? and n = 1) or (not left? and n = -1) =>
plusMinus = 1 => \(-\pi()/(2 :: FE)\)
\(\pi()\)
plusMinus = 1 => \(\pi()/(2 :: FE)\)
0
atan coef
[(cc :: UPS) + integrate(plusMinus * differentiate(ups)/(1 + ups*ups))]

applyIfCan(fcn, arg, fcnName, posCheck?, atanFlag) ==
-- converts fcn(arg) to a power series
(ups := exprToUPS(arg, posCheck?, atanFlag)) case %problem => ups
ans := fcn(ups.%series)
ans case "failed" => stateProblem(fcnName,"essential singularity")
[ans :: UPS]

tranToUPS(ker, arg, posCheck?, atanFlag) ==
-- converts ker to a power series for certain functions
-- in trig or hyperbolic trig categories
is?(ker,"sin" :: SY) =>
applyIfCan(sinIfCan, arg, "sin", posCheck?, atanFlag)
is?(ker,"cos" :: SY) =>
applyIfCan(cosIfCan, arg, "cos", posCheck?, atanFlag)
is?(ker,"tan" :: SY) =>
applyIfCan(tanIfCan, arg, "tan", posCheck?, atanFlag)
is?(ker,"cot" :: SY) =>
applyIfCan(cotIfCan, arg, "cot", posCheck?, atanFlag)
is?(ker,"sec" :: SY) =>
applyIfCan(secIfCan, arg, "sec", posCheck?, atanFlag)
is?(ker,"csc" :: SY) =>
applyIfCan(cscIfCan, arg, "csc", posCheck?, atanFlag)
is?(ker,"asin" :: SY) =>
applyIfCan(asinIfCan, arg, "asin", posCheck?, atanFlag)
is?(ker,"acos" :: SY) =>
applyIfCan(acosIfCan, arg, "acos", posCheck?, atanFlag)
is?(ker,"atan" :: SY) =>
atanIfCanToUPS(arg, posCheck?, atanFlag, 1)
is?(ker,"acot" :: SY) =>
atanIfCanToUPS(arg, posCheck?, atanFlag, -1)
is?(ker,"asec" :: SY) =>
applyIfCan(asecIfCan, arg, "asec", posCheck?, atanFlag)
is?(ker,"acsc" :: SY) =>
applyIfCan(acscIfCan, arg, "acsc", posCheck?, atanFlag)
is?(ker,"sinh" :: SY) =>
applyIfCan(sinhIfCan, arg, "sinh", posCheck?, atanFlag)
is?(ker,"cosh" :: SY) =>
applyIfCan(coshIfCan, arg, "cosh", posCheck?, atanFlag)
is?(ker,"tanh" :: SY) =>
applyIfCan(tanhIfCan, arg, "tanh", posCheck?, atanFlag)
is?(ker,"coth" :: SY) =>
applyIfCan(cothIfCan, arg, "coth", posCheck?, atanFlag)
is?(ker,"sech" :: SY) =>
applyIfCan(sechIfCan, arg, "sech", posCheck?, atanFlag)
is?(ker,"csch" :: SY) =>
    applyIfCan(cschIfCan,arg,"csch",posCheck?,atanFlag)

is?(ker,"asinh" :: SY) =>
    applyIfCan(asinhIfCan,arg,"asinh",posCheck?,atanFlag)

is?(ker,"acosh" :: SY) =>
    applyIfCan(acoshIfCan,arg,"acosh",posCheck?,atanFlag)

is?(ker,"atanh" :: SY) =>
    applyIfCan(atanhIfCan,arg,"atanh",posCheck?,atanFlag)

is?(ker,"acoth" :: SY) =>
    applyIfCan(acothIfCan,arg,"acoth",posCheck?,atanFlag)

is?(ker,"asech" :: SY) =>
    applyIfCan(asechIfCan,arg,"asech",posCheck?,atanFlag)

is?(ker,"acsch" :: SY) =>
    applyIfCan(acschIfCan,arg,"acsch",posCheck?,atanFlag)

stateProblem(string name ker,"unknown kernel")

powToUPS(args,posCheck?,atanFlag) ==
-- converts a power f(x) ** g(x) to a power series
(logBase := logToUPS(first args,posCheck?,atanFlag)) case %problem =>
    logBase
(expon := exprToUPS(second args,posCheck?,atanFlag)) case %problem =>
    expon
ans := expIfCan((expon.%series) * (logBase.%series))
ans case "failed" => stateProblem("exp","essential singularity")
[ans :: UPS]

-- Generalized power series: power series in x, where log(x) and
-- bounded functions of x are allowed to appear in the coefficients
-- of the series. Used for evaluating REAL limits at x = 0.

newElem f ==
-- rewrites a functional expression; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
    smpElem(numer f) / smpElem(denom f)

smpElem p == map(k2Elem,(x1:R):FE +-> x1::FE,p)$PCL

k2Elem k ==
-- rewrites a kernel; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
    null(args := [newElem a for a in argument k]) => k::FE
    iez := inv(ez := exp(z := first args))
    sinz := sin z; cosz := cos z
    is?(k,"tan" :: Symbol) => sinz / cosz
    is?(k,"cot" :: Symbol) => cosz / sinz
    is?(k,"sec" :: Symbol) => inv cosz
    is?(k,"csc" :: Symbol) => inv sinz
    is?(k,"sinh" :: Symbol) => (ez - iez) / (2 :: FE)
is?(k,"cosh" :: Symbol) => (ez + iez) / (2 :: FE)
is?(k,"tanh" :: Symbol) => (ez - iez) / (ez + iez)
is?(k,"coth" :: Symbol) => (ez + iez) / (ez - iez)
is?(k,"sech" :: Symbol) => 2 * inv(ez + iez)
is?(k,"csch" :: Symbol) => 2 * inv(ez - iez)

(operator k) args

CONTFCNS : L S := ["sin","cos","atan","acot","exp","asinh"]
-- functions which are defined and continuous at all real numbers

BDDFCNS : L S := ["sin","cos","atan","acot"]
-- functions which are bounded on the reals

contOnReals? fcn == member?(fcn,CONTFCNS)
bddOnReals? fcn == member?(fcn,BDDFCNS)

exprToGenUPS(fcn,posCheck?,atanFlag) ==
-- converts a functional expression to a generalized power
-- series; "generalized" means that log(x) and bounded functions
-- of x are allowed to appear in the coefficients of the series
iExprToGenUPS(newElem fcn,posCheck?,atanFlag)

iExprToGenUPS(fcn,posCheck?,atanFlag) ==
-- converts a functional expression to a generalized power
-- series without first normalizing the expression
--!! The following line is commented out so that expressions of
--!! the form a**b will be normalized to exp(b * log(a)) even if
--!! 'a' and 'b' do not involve the limiting variable 'x'.
--!! - cjw 1 Dec 94
--not member?(x,variables fcn) => [monomial(fcn,0)]
(poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
[polyToUPS univariate(poly :: POL,x)]
(sum := isPlus fcn) case L(FE) =>
listToUPS(sum :: L(FE),iExprToGenUPS,posCheck?,atanFlag,0,
(y1,y2) +-> y1 + y2)
(prod := isTimes fcn) case L(FE) =>
listToUPS(prod :: L(FE),iExprToGenUPS,posCheck?,atanFlag,1,
(y1,y2) +-> y1 * y2)
(expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
power := expt :: Record(val:FE,exponent:I)
powerToGenUPS(power,val:power.exponent,posCheck?,atanFlag)
(ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
kernelToGenUPS(ker :: K,posCheck?,atanFlag)
error "exprToGenUPS: neither a sum, product, power, nor kernel"

opsInvolvingX fcn ==
opList := [op for k in tower fcn | unary?(op := operator k) _
and member?(x,variables first argument k)]
removeDuplicates opList
opInOpList?(name,opList) ==
  for op in opList repeat
    is?(op,name) => return true
  false

exponential? fcn ==
  -- is 'fcn' of the form exp(f)?
  (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
    is?(ker :: K,"exp" :: Symbol)
  false

productOfNonZeroes? fcn ==
  -- is 'fcn' a product of non-zero terms, where 'non-zero'
  -- means an exponential or a function not involving x
  exponential? fcn => true
  (prod := isTimes fcn) case "failed" => false
  for term in (prod :: L(FE)) repeat
    (not exponential? term) and member?(x,variables term) =>
      return false
  true

powerToGenUPS(fcn,n,posCheck?,atanFlag) ==
  -- converts an integral power to a generalized power series
  -- if n < 0 and the lowest order coefficient of the series
  -- involves x, we are careful about inverting this coefficient
  -- the coefficient is inverted only if
  -- (a) the only function involving x is 'log', or
  -- (b) the lowest order coefficient is a product of exponentials
  -- and functions not involving x
  (b := exprToGenUPS(fcn,posCheck?,atanFlag)) case %problem => b
  n > 0 => [(b.%series) ** n]
  n < 0 =>
    ups := b.%series; deg := degree ups
    if (coef := coefficient(ups,deg)) = 0 then
      deg := order(ups,deg + ZEROCOUNT :: Expon)
      (coef := coefficient(ups,deg)) = 0 =>
        error "inverse of series with many leading zero coefficients"
    xOpList := opsInvolvingX coef
    -- only function involving x is 'log'
    (null xOpList) => [ups ** n]
    (null rest xOpList and is?(first xOpList,"log" :: SY)) => [ups ** n]
    -- lowest order coefficient is a product of exponentials and
    -- functions not involving x
    productOfNonZeroes? coef => [ups ** n]
    stateProblem("inv","lowest order coefficient involves x")
  true

kernelToGenUPS(ker,posCheck?,atanFlag) ==
  -- converts a kernel to a generalized power series
  (sym := symbolIfCan(ker)) case Symbol =>
\[(\text{sym} :: \text{Symbol}) = x \Rightarrow [\text{monomial}(1,1)]\]
\[\text{monomial}(\text{ker} :: \text{FE},0)\]
\[\text{empty?}(\text{args} := \text{argument ker}) \Rightarrow [\text{monomial}(\text{ker} :: \text{FE},0)]\]
\[\text{empty? rest args} =\]
\[\text{arg} := \text{first args}\]
\[\text{is?(ker,\text{"abs"} :: \text{Symbol}) =}\]
\[\text{n}th\text{RootToGenUPS}(\text{arg}*\text{arg},2,\text{posCheck?},\text{atanFlag})\]
\[\text{is?(ker,\text{"%paren"} :: \text{Symbol}) \Rightarrow \text{iExprToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag})}\]
\[\text{is?(ker,\text{"log"} :: \text{Symbol}) \Rightarrow \text{logToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag})}\]
\[\text{is?(ker,\text{"exp"} :: \text{Symbol}) \Rightarrow \text{expToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag})}\]
\[\text{tranToGenUPS}(\text{ker},\text{arg},\text{posCheck?},\text{atanFlag})\]
\[\text{is?(ker,\text{"%power"} :: \text{Symbol}) \Rightarrow \text{powToGenUPS}(\text{args},\text{posCheck?},\text{atanFlag})}\]
\[\text{is?(ker,\text{"nthRoot"} :: \text{Symbol}) =}\]
\[\text{n} := \text{retract}(\text{second args})@I\]
\[\text{n}th\text{RootToGenUPS}(\text{first args},\text{n} :: \text{NNI},\text{posCheck?},\text{atanFlag})\]
\[\text{stateProblem(\text{string name ker},\text{"unknown kernel"})}\]

\[\text{n}th\text{RootToGenUPS}(\text{arg},\text{n},\text{posCheck?},\text{atanFlag}) =\]
\[\text{-- convert an nth root to a power series}\]
\[\text{-- used for computing right hand limits, so the series may have}\]
\[\text{-- non-zero order, but may not have a negative leading coefficient}\]
\[\text{-- when n is even}\]
\[(\text{result} := \text{iExprToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag})) \text{ case } \%\text{problem} =\]
\[\text{result}\]
\[\text{ans} := \text{carefulNthRootIfCan}(\text{result}.%\text{series},\text{n},\text{posCheck?},\text{true})\]
\[\text{ans case } \%\text{problem} =\]
\[\text{ans [%\text{series}}\]

\[\text{logToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag}) =\]
\[\text{-- converts a logarithm log(f(x)) to a generalized power series}\]
\[(\text{result} := \text{iExprToGenUPS}(\text{arg},\text{posCheck?},\text{atanFlag})) \text{ case } \%\text{problem} =\]
\[\text{result}\]
\[\text{ups} := \text{result}.%\text{series}; \text{deg} := \text{degree ups}\]
\[\text{if } (\text{coef} := \text{coefficient}(\text{ups},\text{deg})) = 0 \text{ then}\]
\[\text{deg} := \text{order}(\text{ups},\text{deg} + \text{ZEROCOUNT} :: \text{Expon})\]
\[\text{if } (\text{coef} := \text{coefficient}(\text{ups},\text{deg})) = 0 \Rightarrow\]
\[\text{error "log of series with many leading zero coefficients"}\]
\[\text{if } \text{`posCheck?' is true, we do not allow logs of negative numbers}\]
\[\text{if posCheck? then}\]
\[\text{if } ((\text{signum} := \text{sign}(\text{coef})$\text{SIGNEF}) \text{ case I}) \text{ then}\]
\[\text{(signum :: I) = -1} =\]
\[\text{return stateProblem("log","negative leading coefficient")}\]
\[\text{-- create logarithmic term, avoiding log's of negative rationals}\]
\[\text{lt} := \text{monomial}(\text{coef},\text{deg})$\text{UPS}; \text{cen} := \text{center lt}\]
\[\text{-- check to see if lowest order coefficient is a negative rational}\]
\[\text{negRat?} := \text{Boolean} :=\]
\[\{(\text{rat} := \text{ratIfCan coef} \text{ case RN}) =\]
\[\text{(rat :: RN) < 0} =\]
\[\text{true}\]
\[\text{false}\]
\[\text{false}\]
PACKAGE FS2UPS FUNCTIONSPACETOUNIVARIATEPOWERSERIES
logTerm : FE :=
mon : FE := (x :: FE) - (cen :: FE)
pow : FE := mon ** (deg :: FE)
negRat? => log(coef * pow)
term1 : FE := (deg :: FE) * log(mon)
log(coef) + term1
[monomial(logTerm,0) + log(ups/lt)]
expToGenUPS(arg,posCheck?,atanFlag) ==
-- converts an exponential exp(f(x)) to a generalized
-- power series
(ups := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem => ups
expGenUPS(ups.%series,posCheck?,atanFlag)
expGenUPS(ups,posCheck?,atanFlag) ==
-- computes the exponential of a generalized power series.
-- If the series has order zero and the constant term a0 of the
-- series involves x, the function tries to expand exp(a0) as
-- a power series.
(deg := order(ups,1)) < 0 =>
stateProblem("exp","essential singularity")
deg > 0 => [exp ups]
lc := coefficient(ups,0); xOpList := opsInvolvingX lc
not opInOpList?("log" :: SY,xOpList) => [exp ups]
-- try to fix exp(lc) if necessary
expCoef :=
normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
opInOpList?("log" :: SY,opsInvolvingX expCoef) =>
stateProblem("exp","logs in constant coefficient")
result := exprToGenUPS(expCoef,posCheck?,atanFlag)
result case %problem => result
[(result.%series) * exp(ups - monomial(lc,0))]
atancotToGenUPS(fe,arg,posCheck?,atanFlag,plusMinus) ==
-- converts atan(f(x)) to a generalized power series
(result := exprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
trouble := result.%problem
trouble.prob = "essential singularity" => [monomial(fe,0)$UPS]
result
ups := result.%series; coef := coefficient(ups,0)
-- series involves complex numbers
(ord := order(ups,0)) = 0 and coef * coef = -1 =>
y := differentiate(ups)/(1 + ups*ups)
yCoef := coefficient(y,-1)
[monomial(log yCoef,0) + integrate(y - monomial(yCoef,-1)$UPS)]
cc : FE :=
ord < 0 =>
atanFlag = "complex" =>
return stateProblem("atan","essential singularity")
(rn := ratIfCan(ord :: FE)) case "failed" =>

839


chapter 7. chapter f

-- this condition usually won't occur because exponents will
-- be integers or rational numbers
return stateProblem("atan","branch problem")

if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
-- expansions to the left and right of zero have different
-- constant coefficients
return stateProblem("atan","branch problem")

lc := coefficient(ups,ord)
(signum := sign(lc)$SIGNEF) case "failed" =>
-- can't determine sign
atanFlag = "just do it" =>
  plusMinus = 1 => pi()/(2 :: FE)
  0
posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
plusMinus = 1 => posNegPi2
pi()/(2 :: FE) - posNegPi2
--return stateProblem("atan","branch problem")

left? : B := atanFlag = "real: left side"; n := signum :: Integer
(left? and n = 1) or (not left? and n = -1) =>
  plusMinus = 1 => -pi()/(2 :: FE)
  pi()
  plusMinus = 1 => pi()/(2 :: FE)
  0
atan coef

[(cc :: UPS) + integrate(differentiate(ups)/(1 + ups*ups))]

"atan coef" [(cc :: UPS) + integrate(differentiate(ups)/(1 + ups*ups))]

\[
\text{genUPSApplyIfCan}(\text{fcn}, \text{arg}, \text{fcnName}, \text{posCheck}, \text{atanFlag}) =
\]
-- converts fcn(arg) to a generalized power series
(series := iExprToGenUPS(\text{arg}, \text{posCheck}, \text{atanFlag})) case %problem =>

series

ups := series.%series
(deg := order(ups,1)) < 0 =>
  stateProblem(fcnName,"essential singularity")

deg > 0 => \[\text{fcn}(\text{ups}) :: UPS\]

lc := coefficient(ups,0); xOpList := opsInvolvingX lc
null xOpList => \[\text{fcn}(\text{ups}) :: UPS\]

opInOpList(?"log" :: SY,xOpList) =>
  stateProblem(fcnName,"logs in constant coefficient")

contOnReals? fcnName => \[\text{fcn}(\text{ups}) :: UPS\]

stateProblem(fcnName,\"x in constant coefficient\")

\[
\text{applyBddIfCan}(\text{fe}, \text{fcn}, \text{arg}, \text{fcnName}, \text{posCheck}, \text{atanFlag}) =
\]
-- converts fcn(arg) to a generalized power series, where the
-- function fcn is bounded for real values
-- if fcn(arg) has an essential singularity as a complex
-- function, we return fcn(arg) as a monomial of degree 0
(ups := iExprToGenUPS(\text{arg}, \text{posCheck}, \text{atanFlag})) case %problem =>

trouble := ups.%problem

trouble.prob = "essential singularity" => \[\text{monomial}(\text{fe},0)$UPS\]

ups
 tranToGenUPS(ker, arg, posCheck?, atanFlag) ==
-- converts op(arg) to a power series for certain functions
-- op in trig or hyperbolic trig categories
-- N.B. when this function is called, 'k2elem' will have been
-- applied, so the following functions cannot appear:
-- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
is?(ker, "sin" :: SY) =>
  applyBddIfCan(ker :: FE, sinIfCan, arg, "sin", posCheck?, atanFlag)
is?(ker, "cos" :: SY) =>
  applyBddIfCan(ker :: FE, cosIfCan, arg, "cos", posCheck?, atanFlag)
is?(ker, "asin" :: SY) =>
  genUPSApplyIfCan(asinIfCan, arg, "asin", posCheck?, atanFlag)
is?(ker, "acos" :: SY) =>
  genUPSApplyIfCan(acosIfCan, arg, "acos", posCheck?, atanFlag)
is?(ker, "atan" :: SY) =>
  atancotToGenUPS(ker :: FE, arg, posCheck?, atanFlag, 1)
is?(ker, "acot" :: SY) =>
  atancotToGenUPS(ker :: FE, arg, posCheck?, atanFlag, -1)
is?(ker, "asec" :: SY) =>
  genUPSApplyIfCan(asecIfCan, arg, "asec", posCheck?, atanFlag)
is?(ker, "acsc" :: SY) =>
  genUPSApplyIfCan(acscIfCan, arg, "acsc", posCheck?, atanFlag)
is?(ker, "asinh" :: SY) =>
  genUPSApplyIfCan(asinhIfCan, arg, "asinh", posCheck?, atanFlag)
is?(ker, "acosh" :: SY) =>
  genUPSApplyIfCan(acoshIfCan, arg, "acosh", posCheck?, atanFlag)
is?(ker, "atanh" :: SY) =>
  genUPSApplyIfCan(atanhIfCan, arg, "atanh", posCheck?, atanFlag)
is?(ker, "atanh" :: SY) =>
  genUPSApplyIfCan(acothIfCan, arg, "acoth", posCheck?, atanFlag)
is?(ker, "asech" :: SY) =>
  genUPSApplyIfCan(asechIfCan, arg, "asech", posCheck?, atanFlag)
is?(ker, "acsch" :: SY) =>
  genUPSApplyIfCan(acschIfCan, arg, "acsch", posCheck?, atanFlag)
stateProblem(string name ker, "unknown kernel")

 powToGenUPS(args, posCheck?, atanFlag) ==
-- converts a power f(x) ** g(x) to a generalized power series
{logBase := logToGenUPS(first args, posCheck?, atanFlag)} case %problem =>
  logBase
expon := iExprToGenUPS(second args, posCheck?, atanFlag)
expon case %problem => expon
expGenUPS((expon.%series) * (logBase.%series), posCheck?, atanFlag)
package FSUPFACT FunctionSpaceUnivariatePolynomialFactor

--- FunctionSpaceUnivariatePolynomialFactor.input ---

)set break resume
/sys rm -f FunctionSpaceUnivariatePolynomialFactor.output
/spool FunctionSpaceUnivariatePolynomialFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceUnivariatePolynomialFactor
--E 1

)spool
)lisp (bye)

--- FunctionSpaceUnivariatePolynomialFactor.help ---

====================================================================
FunctionSpaceUnivariatePolynomialFactor examples
====================================================================

This package is used internally by IR2F

See Also:
  o )show FunctionSpaceUnivariatePolynomialFactor
FSUPFACT

Exports:
anfactor ffactor qfactor UP2ifCan

---

)abbrev package FSUPFACT FunctionSpaceUnivariatePolynomialFactor
++ Author: Manuel Bronstein
++ Date Created: 12 May 1988
++ Date Last Updated: 22 September 1993
++ Description:
++ This package is used internally by IR2F

FunctionSpaceUnivariatePolynomialFactor(R, F, UP):
Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
  F : FunctionSpace R
  UP: UnivariatePolynomialCategory F

  Q ==> Fraction Integer
  K ==> Kernel F
  AN ==> AlgebraicNumber
  PQ ==> SparseMultivariatePolynomial(Q, K)
  PR ==> SparseMultivariatePolynomial(R, K)
  UPQ ==> SparseUnivariatePolynomial Q
  UPA ==> SparseUnivariatePolynomial AN
  FR ==> Factored UP
  FRQ ==> Factored UPQ
  FRA ==> Factored UPA
CHAPTER 7. CHAPTER F

Exports ==> with
ffactor: UP -> FR
  ++ ffactor(p) tries to factor a univariate polynomial p over F
qfactor: UP -> Union(FRQ, "failed")
  ++ qfactor(p) tries to factor p over fractions of integers,
  ++ returning "failed" if it cannot
UP2ifCan: UP -> Union(overq: UPQ, overan: UPA, failed: Boolean)
  ++ UP2ifCan(x) should be local but conditional.
if F has RetractableTo AN then
  anfactor: UP -> Union(FRA, "failed")
  ++ anfactor(p) tries to factor p over algebraic numbers,
  ++ returning "failed" if it cannot

Implementation ==> add
import AlgFactor(UPA)
import RationalFactorize(UPQ)

P2QifCan : PR -> Union(PQ, "failed")
UPQ2UP : (SparseUnivariatePolynomial PQ, F) -> UP
PQ2F : (PQ, F) -> F
ffactor0 : UP -> FR

dummy := kernel(new()$Symbol)$K

if F has RetractableTo AN then
  UPAN2F: UPA -> UP
  UPQ2AN: UPQ -> UPA

UPAN2F p ==
  map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(AN,UPA,F,UP)

UPQ2AN p ==
  map(x+->x::AN, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,AN,UPA)

ffactor p ==
  (pq := anfactor p) case FRA =>
    map(UPAN2F, pq::FRA)$FactoredFunctions2(UPA, UP)
  ffactor0 p

anfactor p ==
  (q := UP2ifCan p) case overq =>
    map(UPQ2AN, factor(q.overq))$FactoredFunctions2(UPQ, UPA)
  q case overan => factor(q.overan)
  "failed"

UP2ifCan p ==
  ansq := 0$UPQ ; ansa := 0$UPA
goforq? := true
while p ^= 0 repeat
  if goforq? then
rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
if rq case Q then
   ansq := ansq + monomial(rq::Q, degree p)
   ansa := ansa + monomial(rq::Q::AN, degree p)
else
   goforq? := false
   ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
   if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
   else return [true]
else
   ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
   if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
   else return [true]
p := reductum p
goforq? => [ansq]
[ansa]
else
 UPQ2F: UPQ -> UP

UPQ2F p ==
 map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

ffactor p ==
 (pq := qfactor p) case FRQ =>
    map(UPQ2F, pq::FRQ)$FactoredFunctions2(UPQ, UP)
ffactor0 p

UP2ifCan p ==
 ansq := 0$UPQ
while p ^= 0 repeat
   rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
   if rq case Q then ansq := ansq + monomial(rq::Q, degree p)
   else return [true]
p := reductum p
[ansq]

ffactor0 p ==
 smp := numer(ep := p(dummy::F))
 (q := P2ifCan smp) case "failed" => p::FR
 map(x+->UPQ2UP(univariate(x, dummy), denom(ep)::F), factor(q::PQ )$MRationalFactorize(IndexedExponents K, K, Integer, PQ))$FactoredFunctions2(PQ, UP)

UPQ2UP(p, d) ==
 map(x+->PQ2F(x, d), p)$UnivariatePolynomialCategoryFunctions2(PQ, SparseUnivariatePolynomial PQ, F, UP)

PQ2F(p, d) ==
 map((x:K):F+->x::F, (y:Q):F+->y::F, p)
$\text{PolynomialCategoryLifting}(\text{IndexedExponents } K, K, Q, PQ, F) / d$

\text{qfactor } p == 
(q := \text{UP2ifCan } p) \text{ case overq => factor}(q.\text{overq})
"failed"

\text{P2QifCan } p ==
\text{and/}[\text{retractIfCan}(c::F)\text{\@Union}(Q, "failed") \text{ case } Q
  \text{ for } c \text{ in coefficients } p] \Rightarrow
  \text{map}(x \mapsto x::PQ, y \mapsto \text{retract}(y::F)\text{\@Q} :: PQ, p)_-
$\text{PolynomialCategoryLifting}(\text{IndexedExponents } K,K,R,PR,PQ)
"failed"

— FSUPFACT.dotabb —

"FSUPFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSUPFACT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FSUPFACT" -> "FS"
"FSUPFACT" -> "ACF"

—
Chapter 8

Chapter G

package GALFACTU GaloisGroupFactorizationUtilities

--- GaloisGroupFactorizationUtilities.input ---

)set break resume
)sys rm -f GaloisGroupFactorizationUtilities.output
)spool GaloisGroupFactorizationUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupFactorizationUtilities
--E 1

)spool
)lisp (bye)

---

--- GaloisGroupFactorizationUtilities.help ---

====================================================================
GaloisGroupFactorizationUtilities examples
====================================================================

GaloisGroupFactorizationUtilities provides functions that will be used by the factorizer.

See Also:

847
GaloisGroupFactorizationUtilities (GALFACTU)

Exports:
beauzamyBound  bombieriNorm  height  infinityNorm  length
norm  quadraticNorm  rootBound  singleFactorBound

--- package GALFACTU GaloisGroupFactorizationUtilities ---

)abbrev package GALFACTU GaloisGroupFactorizationUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 19 October 1995
++ References:
++ [1] Bernard Beauzamy, Products of polynomials and a priori estimates for
++ coefficients in polynomial decompositions: a sharp result,
++ J. Symbolic Computation (1992) 13, 463-472
++ [2] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri’s Norms: I. The Largest Factor,
++ J. Symbolic Computation (1993) 16, 115-130
++ [3] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri’s Norms: II. The Smallest Factor,
++ J. Symbolic Computation (1993) 16, 131-145
++ [4] Maurice Mignotte, Some Useful Bounds,
++ Computing, Suppl. 4, 259-263 (1982), Springer-Verlag
++ Algorithms) 1st edition, 2nd printing, Addison-Wesley 1971, p. 397-398
++ Factorization: Sharp Bounds, Efficient Algorithms,
++ J. Symbolic Computation (1993) 15, 393-413

Description:
\texttt{GaloisGroupFactorizationUtilities} provides functions that will be used by the factorizer.

\texttt{GaloisGroupFactorizationUtilities}(R,UP,F): Exports == Implementation where
\texttt{R} : Ring
\texttt{UP} : UnivariatePolynomialCategory R
\texttt{F} : Join(FloatingPointSystem,RetractableTo(R),Field,TranscendentalFunctionCategory,ElementaryFunctionCategory)
\texttt{N} ==> NonNegativeInteger
\texttt{P} ==> PositiveInteger
\texttt{Z} ==> Integer

Exports ==>
\begin{verbatim}
beauzamyBound: UP -> Z -- See [1]
  ++ beauzamyBound(p) returns a bound on the larger coefficient of any
  ++ factor of p.
bombieriNorm: UP -> F -- See [1]
  ++ bombieriNorm(p) returns quadratic Bombieri's norm of p.
bombieriNorm: (UP,P) -> F -- See [2] and [3]
  ++ bombieriNorm(p,n) returns the nth Bombieri's norm of p.
  ++ rootBound(p) returns a bound on the largest norm of the complex roots
  ++ of p.
singleFactorBound: (UP,N) -> Z -- See [6]
  ++ singleFactorBound(p,r) returns a bound on the infinite norm of
  ++ the factor of p with smallest Bombieri's norm. r is a lower bound
  ++ for the number of factors of p. p shall be of degree higher or equal
  ++ to 2.
singleFactorBound: UP -> Z -- See [6]
  ++ singleFactorBound(p,r) returns a bound on the infinite norm of
  ++ the factor of p with smallest Bombieri's norm. p shall be of degree
  ++ higher or equal to 2.
norm: (UP,P) -> F
  ++ norm(f,p) returns the lp norm of the polynomial f.
quadraticNorm: UP -> F
  ++ quadraticNorm(f) returns the l2 norm of the polynomial f.
infinityNorm: UP -> F
  ++ infinityNorm(f) returns the maximal absolute value of the coefficients
  ++ of the polynomial f.
height: UP -> F
  ++ height(p) returns the maximal absolute value of the coefficients of
  ++ the polynomial p.
length: UP -> F
  ++ length(p) returns the sum of the absolute values of the coefficients
  ++ of the polynomial p.
\end{verbatim}

Implementation ==> add
import GaloisGroupUtilities(F)

height(p:UP):F == infinityNorm(p)

length(p:UP):F == norm(p,1)

norm(f:UP,p:P):F ==
  n : F := 0
  for c in coefficients f repeat
    n := n+abs(c::F)**p
  nthRoot(n,p::N)

quadraticNorm(f:UP):F == norm(f,2)

infinityNorm(f:UP):F ==
  n : F := 0
  for c in coefficients f repeat
    n := max(n,c::F)
  n

  n : N := degree p
  r := max(2,r)
  n < r => error "singleFactorBound: Bad arguments."
  nf : F := n :: F
  num : F := nthRoot(bombieriNorm(p),r)
  if F has Gamma: F -> F then
    num := num*nthRoot(Gamma(nf+1$F),2*r)
    den : F := Gamma((2*r)::F)+1$F)
  else
    num := num*(2::F)**(5/8+n/2)*exp(1$F/(4*nf))
    den : F := (pi()$F*nf)**(3/8)
  safeFloor( num/den )

singleFactorBound(p:UP,r:N):Z == singleFactorBound(p,2) -- See [6]

  n := degree p
  zero? n => 0
  lc := abs(leadingCoefficient(p)::F)
  b1 : F := 0 -- Mignotte
  b2 : F := 0 -- Knuth
  b3 : F := 0 -- Zassenhaus in [5]
  b4 : F := 0 -- Cauchy in [7]
  c : F := 0
  cl : F := 0
  for i in 1..n repeat
    c := abs(coefficient(p,(n-i)::N)::F)
    b1 := max(b1,c)
\[ cl := \frac{c}{l_c} \]
\[ b_2 := \max(b_2, \text{nthRoot}(cl, i)) \]
\[ b_3 := \max(b_3, \text{nthRoot}(cl/pascalTriangle(n, i), i)) \]
\[ b_4 := \max(b_4, \text{nthRoot}(n \cdot cl, i)) \]
\[ \min(1 + \text{safeCeiling}(b_1/l_c), \min(\text{safeCeiling}(2 \cdot b_2), \min(\text{safeCeiling}(b_3/\text{nRoot}(2::F, n)-1), \text{safeCeiling}(b_4)))) \]

\textbf{beauzamyBound}(f:UP):Z == -- See [1]
\[
d := \text{degree } f
\]
\[
\text{zero? } d \Rightarrow \text{safeFloor bombieriNorm } f
\]
\[
\text{safeFloor}( (\text{bombieriNorm}(f)*(3::F)**(3/4+d/2))/ (2*\text{sqrt}(\pi()^F*(d::F))))
\]

\[
d := \text{degree } f
\]
\[
b := \text{abs(coefficient}(f,0)::F)
\]
\[
\text{if zero? } d \text{ then return } b
\]
\[
\text{else } b := b^p
\]
\[
b := b \cdot \text{abs(leadingCoefficient}(f)::F)^p
\]
\[
dd := (d-1) \text{ quo } 2
\]
\[
\text{for } i \text{ in } 1..\text{dd repeat}
\]
\[
b := b^p + \text{abs(coefficient}(f,i)::F)^p + \text{abs(coefficient}(f, (d-i)::N)::F)^p
\]
\[
/pascalTriangle(d,i)
\]
\[
\text{if even? } d \text{ then}
\]
\[
\text{dd := dd+1}
\]
\[
b := b^p + \text{abs(coefficient}(f, dd::N)::F)^p/pascalTriangle(d,dd)
\]
\[
nthRoot(b,p::N)
\]

\textbf{bombieriNorm}(f:UP):F == \text{bombieriNorm}(f,2) -- See [1]

---

---

package GALFACT GaloisGroupFactorizer

---

GaloisGroupFactorizer.input ---

)set break resume
GaloisGroupFactorizer (GALFACT)

Exports:
<table>
<thead>
<tr>
<th>PACKAGE GALFACT GALOISGROUPFACTORIZER</th>
</tr>
</thead>
</table>

btwFact  
degreePartition  
eisensteinIrreducible?  
factor  
factorSquareFree  
henselFact  
makeFR  
modularFactor  
numberOfFactors  
stopMusserTrials  
tryFunctionalDecomposition  
tryFunctionalDecomposition?  
useEisensteinCriterion?  
useSingleFactorBound  
useSingleFactorBound?  

--- package GALFACT GaloisGroupFactorizer ---

)abbrev package GALFACT GaloisGroupFactorizer  
++ Author: Frederic Lehobey  
++ Date Created: 28 June 1994  
++ Date Last Updated: 11 July 1997  
++ References:  
++ Factorization: Sharp Bounds, Efficient Algorithms,  
++ J. Symbolic Computation (1993) 15, 393-413  
++ [2] John Brillhart, Note on Irreducibility Testing,  
++ [3] David R. Musser, On the Efficiency of a Polynomial Irreducibility Test,  
++ Description:  
++ \spadtype{GaloisGroupFactorizer} provides functions to factor resolvents.  
-- improvements to do :  
-- + reformulate the lifting problem in completeFactor -- See [1] (hard)  
-- + implement algorithm RC -- See [1] (easy)  
-- + use Dedekind's criterion to prove sometimes irreducibility (easy)  
-- or even to improve early detection of true factors (hard)  
-- + replace Sets by Bits

GaloisGroupFactorizer(UP):Exports == Implementation where  
  Z ==> Integer  
  UP: UnivariatePolynomialCategory Z  
  N ==> NonNegativeInteger  
  P ==> PositiveInteger  
  CYC ==> CyclotomicPolynomialPackage()  
  SUPZ ==> SparseUnivariatePolynomial Z

ParFact ==> Record(irr: UP, pow: Z)  
FinalFact ==> Record(contp: Z, factors: List ParFact)  
DDRecord ==> Record(factor: UP, degree: Z) -- a Distinct-Degree factor  
DDLlist ==> List DDRecord  
MFact ==> Record(prime: Z,factors: List UP) -- Modular Factors  
LR ==> Record(left: UP, right: UP) -- Functional decomposition

Exports == with  
  makeFR: FinalFact -> Factored UP  
    ++ makeFR(flist) turns the final factorization of henselFact into a  
    ++ \spadtype{Factored} object.  
  degreePartition: DDLlist -> Multiset N  
    ++ degreePartition(ddfactorization) returns the degree partition of  
    ++ the polynomial f modulo p where ddfactorization is the distinct
++ degree factorization of f computed by ddFact for some prime p.

musserTrials:: () -> P
++ musserTrials() returns the number of primes that are tried in
++ \spadfun{modularFactor}.

musserTrials:: P -> P
++ musserTrials(n) sets to n the number of primes to be tried in
++ \spadfun{modularFactor} and returns the previous value.

stopMusserTrials:: () -> P
++ stopMusserTrials() returns the bound on the number of factors for
++ which \spadfun{modularFactor} stops to look for an other prime. You
++ will have to remember that the step of recombining the extraneous
++ factors may take up to \spad{2**stopMusserTrials()} trials.

stopMusserTrials:: P -> P
++ stopMusserTrials(n) sets to n the bound on the number of factors for
++ which \spadfun{modularFactor} stops to look for an other prime. You
++ will have to remember that the step of recombining the extraneous
++ factors may take up to \spad{2**n} trials. Returns the previous
++ value.

numberOfFactors:: DDList -> N
++ numberOfFactors(ddfactorization) returns the number of factors of
++ the polynomial f modulo p where ddfactorization is the distinct
++ degree factorization of f computed by ddFact for some prime p.

modularFactor:: UP -> MFact
++ modularFactor(f) chooses a "good" prime and returns the factorization
++ of f modulo this prime in a form that may be used by
++ completeHensel. If prime is zero
++ it means that f has been proved to be irreducible over the integers
++ or that f is a unit (i.e. 1 or -1).
++ f shall be primitive (i.e. content(p)=1) and square free (i.e.
++ without repeated factors).

useSingleFactorBound?: () -> Boolean
++ useSingleFactorBound?() returns \spad{true} if algorithm with single
++ factor bound is used for factorization, \spad{false} for algorithm
++ with overall bound.

useSingleFactorBound: Boolean -> Boolean
++ useSingleFactorBound(b) chooses the algorithm to be used by the
++ factorizers: \spad{true} for algorithm with single
++ factor bound, \spad{false} for algorithm with overall bound.
++ Returns the previous value.

useEisensteinCriterion?: () -> Boolean
++ useEisensteinCriterion?() returns \spad{true} if factorizers
++ check Eisenstein's criterion before factoring.

useEisensteinCriterion: Boolean -> Boolean
++ useEisensteinCriterion(b) chooses whether factorizers check
++ Eisenstein's criterion before factoring: \spad{true} for
++ using it, \spad{false} else. Returns the previous value.

eisensteinIrreducible?: UP -> Boolean
++ eisensteinIrreducible?(p) returns \spad{true} if p can be
++ shown to be irreducible by Eisenstein's criterion,
++ \spad{false} is inconclusive.
tryFunctionalDecomposition?: () -> Boolean
++ tryFunctionalDecomposition?() returns \spad{true} if
++ factorizers try functional decomposition of polynomials before
++ factoring them.
tryFunctionalDecomposition: Boolean -> Boolean
++ tryFunctionalDecomposition(b) chooses whether factorizers have
++ to look for functional decomposition of polynomials
++ (\spad{true}) or not (\spad{false}). Returns the previous value.
factor: UP -> Factored UP
++ factor(p) returns the factorization of p over the integers.
factor: (UP,N) -> Factored UP
++ factor(p,r) factorizes the polynomial p using the single factor bound
++ algorithm and knowing that p has at least r factors.
factor: (UP,List N) -> Factored UP
++ factor(p,listOfDegrees) factorizes the polynomial p using the single
++ factor bound algorithm and knowing that p has for possible
++ splitting of its degree listOfDegrees.
factor: (UP,List N,N) -> Factored UP
++ factor(p,listOfDegrees,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that p has for possible
++ splitting of its degree listOfDegrees and that p has at least r
++ factors.
factor: (UP,N,N) -> Factored UP
++ factor(p,d,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that d divides the degree of all
++ factors of p and that p has at least r factors.
factorSquareFree: UP -> Factored UP
++ factorSquareFree(p) returns the factorization of p which is supposed
++ not having any repeated factor (this is not checked).
factorSquareFree: (UP,N) -> Factored UP
++ factorSquareFree(p,r) factorizes the polynomial p using the single
++ factor bound algorithm and knowing that p has at least r factors.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N) -> Factored UP
++ factorSquareFree(p,listOfDegrees) factorizes the polynomial p using
++ the single factor bound algorithm and knowing that p has for possible
++ splitting of its degree listOfDegrees.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N,N) -> Factored UP
++ factorSquareFree(p,listOfDegrees,r) factorizes the polynomial p using
++ the single factor bound algorithm, knowing that p has for possible
++ splitting of its degree listOfDegrees and that p has at least r
++ factors.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,N,N) -> Factored UP
++ factorSquareFree(p,d,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that d divides the degree of all
++ factors of p and that p has at least r factors.
++ f is supposed not having any repeated factor (this is not checked).
factorOfDegree: (P,UP) -> Union(UP,"failed")
++ factorOfDegree(d,p) returns a factor of p of degree d.
factorOfDegree: (P,UP,N) -> Union(UP,"failed")
++ factorOfDegree(d,p,r) returns a factor of p of degree
++ d knowing that p has at least r factors.
factorOfDegree: (P,UP,List N) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees) returns a factor
++ of p of degree d knowing that p has for possible splitting of its
++ degree listOfDegrees.
factorOfDegree: (P,UP,List N,N) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees,r) returns a factor
++ of p of degree d knowing that p has for possible splitting of its
++ degree listOfDegrees, and that p has at least r factors.
factorOfDegree: (P,UP,List N,N,Boolean) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees,r,sqf) returns a
++ factor of p of degree d knowing that p has for possible splitting of
++ its degree listOfDegrees, and that p has at least r factors.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
henselFact: (UP,Boolean) -> FinalFact
++ henselFact(p,sqf) returns the factorization of p, the result
++ is a Record such that \spad{contp=}content p,
++ \spad{factors=}List of irreducible factors of p with exponent.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
btwFact: (UP,Boolean,Set N,N) -> FinalFact
++ btwFact(p,sqf,pd,r) returns the factorization of p, the result
++ is a Record such that \spad{contp=}content p,
++ \spad{factors=}List of irreducible factors of p with exponent.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
++ pd is the \spadtype{Set} of possible degrees. r is a lower bound for
++ the number of factors of p. Please do not use this function in your
++ code because its design may change.

Implementation ==> add

fUnion ==> Union("nil", "sqfr", "irred", "prime")
FFE ==> Record(flg:fUnion, fctr:UP, xpnt:Z) -- Flag-Factor-Exponent
DDFact ==> Record(prime:Z, ddfactors:DDList) -- Distinct Degree Factors
HLR ==> Record(plist:List UP, modulo:Z) -- HenselLift Record

mussertials: P := 5
stopmussertials: P := 8
usesinglefactorbound: Boolean := true
tryfunctionaldecomposition: Boolean := true
useEisensteinCriterion: Boolean := true

useEisensteinCriterion?():Boolean == useEisensteinCriterion

useEisensteinCriterion(b:Boolean):Boolean ==
(useeisenstein criterion, b) := (b, useeisenstein criterion) b

tryFunctionalDecomposition?(): Boolean == tryfunctionaldecomposition

tryFunctionalDecomposition(b: Boolean): Boolean ==
  (tryfunctionaldecomposition, b) := (b, tryfunctionaldecomposition) b

useSingleFactorBound?(): Boolean == usesinglefactorbound

useSingleFactorBound(b: Boolean): Boolean ==
  (usesinglefactorbound, b) := (b, usesinglefactorbound) b

stopMusserTrials(): P == stopmussertrials

stopMusserTrials(n: P): P ==
  (stopmussertrials, n) := (n, stopmussertrials) n

musserTrials(): P == mussertrials

musserTrials(n: P): P ==
  (mussertrials, n) := (n, mussertrials) n

import GaloisGroupFactorizationUtilities(Z, UP, Float)

import GaloisGroupPolynomialUtilities(Z, UP)

import IntegerPrimesPackage(Z)
import IntegerFactorizationPackage(Z)
import ModularDistinctDegreeFactorizer(UP)

eisensteinIrreducible?(f: UP): Boolean ==
  rf := reductum f
c: Z := content rf
zero? c => false
unit? c => false
lc := leadingCoefficient f
tc := lc
while not zero? rf repeat
  tc := leadingCoefficient rf
  rf := reductum rf
  for p in factors(factor c)$Factored(Z) repeat
    -- if (one? p.exponent) and (not zero? (lc rem p.factor)) and
    if (p.exponent = 1) and (not zero? (lc rem p.factor)) and
       (not zero? (tc rem ((p.factor)**2))) then return true
false

numberOfFactors(ddlist:DDList):N ==
n: N := 0
d: Z := 0
for dd in ddlist repeat
  n := n +
  zero? (d := degree(dd.factor)::Z) => 1
  (d quo dd.degree)::N
  n

-- local function, returns the a Set of shifted elements
shiftSet(s:Set N,shift:N):Set N == set [ e+shift for e in parts s ]

-- local function, returns the "reductum" of an Integer (as chain of bits)
reductum(n:Z):Z == n-shift(1,length(n)-1)

-- local function, returns an integer with level lowest bits set to 1
seed(level:Z):Z == shift(1,level)-1

-- local function, returns the next number (as a chain of bit) for
-- factor reconciliation of a given level (which is the number of
-- extraneous factors involved) or "End of level" if not any
nextRecNum(levels:N,level:Z,n:Z):Union("End of level",Z) ==
  if (l := length n)<levels then return(n+shift(1,l-1))
  (n=shift(seed(level),levels-level)) => "End of level"
  b: Z := 1
  while ((l-b) = (lr := length(n := reductum n)))@Boolean repeat b := b+1
  reductum(n)+shift(seed(b+1),lr)

-- local function, return the set of N, 0..n
fullSet(n:N):Set N == set [ i for i in 0..n ]

modularFactor(p:UP):MFact ==
  -- not one? abs(content(p)) =>
  not (abs(content(p)) = 1) =>
  error "modularFactor: the polynomial is not primitive."
  zero? (n := degree p) => [0,[p]]

-- declarations --
cprime: Z := 2
trials: List DDFact := empty()
d: Set N := fullSet(n)
dirred: Set N := set [0,n]
s: Set N := empty()
ddlst: DDList := empty()
degfact: N := 0
nf: N := stopmussertrials+1
i: Z
-- Musser, see [3] --
diffp := differentiate p
for i in 1..mussertrials | nf>stopmussertrials repeat
  -- test 1: cprime divides leading coefficient
  -- test 2: "bad" primes: (in future: use Dedekind's Criterion)
  while (zero? ((leadingCoefficient p) rem cprime)) or
    (not zero? degree gcd(p,diffp,cprime)) repeat
    cprime := nextPrime(cprime)
  ddlist := ddFact(p,cprime)
  -- degree compatibility: See [3] --
  s := set [0]
  for f in ddlist repeat
    degfact := f.degree::N
    if not zero? degfact then
      for j in 1..(degree(f.factor) quo degfact) repeat
        s := union(s, shiftSet(s,degfact))
    trials := cons([cprime,ddlist]$DDFact,trials)
  d := intersect(d, s)
  d = dirred => return [0,[p]] -- p is irreducible
  cprime := nextPrime(cprime)
  nf := numberOfFactors ddlist

  -- choose the one with the smallest number of factors
  choice := first trials
  nfc := numberOfFactors(choice.ddfactors)
  for t in rest trials repeat
    nf := numberOfFactors(t.ddfactors)
    if nf<nfc or ((nf=nfc) and (t.prime>choice.prime)) then
      nfc := nf
      choice := t
  cprime := choice.prime
  -- HenselLift$GHENSEL expects the degree 0 factor first
  [cprime,separateFactors(choice.ddfactors,cprime)]

degreePartition(ddlist:DDList):Multiset N ==
  dp: Multiset N := empty()
  d: N := 0
  dd: N := 0
  for f in ddlist repeat
    zero? (d := degree(f.factor)) => dp := insert!(0,dp)
    dd := f.degree::N
    dp := insert!(dd,dp,d quo dd)
  dp

import GeneralHenselPackage(Z,UP)
import UnivariatePolynomialDecompositionPackage(Z,UP)
import BrillhartTests(UP) -- See [2]

-- local function, finds the factors of f primitive, square-free, with
-- positive leading coefficient and non zero trailing coefficient,
henselfact(f:UP,pdecomp:Boolean):List UP ==
  if brillhartIrreducible? f or
    (useeisensteinCriterion => eisensteinIrreducible? f ; false)
  then return [f]
cf: Union(LR,"failed")
  if pdecomp and tryfunctionaldecomposition then
    cf := monicDecomposeIfCan f
  else
    cf case "failed" =>
      m := modularFactor f
      zero? (cprime := m.prime) => m.factors
      b: P := (2*leadingCoefficient(f)*beauzamyBound(f)) :: P
      completeHensel(f,m.factors,cprime,b)
lrf := cf::LR
      "append"/[ henselfact(g(lrf.right),false) for g in
        henselfact(lrf.left,true) ]

  lc := leadingCoefficient f
f0 := coefficient(f,0)
ltrue: List UP := empty()
found? := true
degf: N := 0
degg: N := 0
g0: Z := 0
g: UP := 0
rg: N := 0
nb: Z := 0
lg: List UP := empty()
b: P := 1
dg: Set N := empty()
llg: HLR := [empty(),0]
levels: N := #lf
level: Z := 1
ic: Union(Z,"End of level") := 0
i: Z := 0
while level<levels repeat
  -- try all possible factors with degree in d
  ic := seed(level)
  while ((not found?) and (ic case Z)) repeat
    i := ic::Z
degg := 0
g0 := 1 -- LC algorithm
    for j in 1..levels repeat
      if bit?(i,j-1) then
degg := degg+degree lf.j

g0 := g0*coefficient(lf.j,0) -- LC algorithm

g0 := symmetricRemainder(lc*g0,pk) -- LC algorithm

if member?(degg,d) and (((lc*f0) exquo g0) case Z) then
    -- LC algorithm
    
g := lc::UP -- build the possible factor -- LC algorithm

    for j in 1..levels repeat if bit?(i,j-1) then g := g*lf.j

    g := primitivePart reduction(g,pk)

    f1 := f exquo g

    if f1 case UP then -- g is a true factor
        found? := true

        -- remove the factors of g from lf

        for j in 1..levels repeat

            if bit?(i,j-1) then
                swap!(lf,j,nb)
            
            nb := nb+1

        end for

        lg := lf

        lf := rest(lf,level::N)

        setrest!(rest(lg,(level-1)::N),empty()$List(UP))

        f := f1::UP

        lc := leadingCoefficient f

        f0 := coefficient(f,0)

        -- is g irreducible?

        dg := select(x+->x <= degg,d)

        if not(dg=set [0,degg]) then -- implies degg >= 2

            rg := max(2,(r+level-levels)::N)

            b := (2*leadingCoefficient(g)*singleFactorBound(g,rg)) :: P

            if b>pk and (not brillhartIrreducible?(g)) and
                (useeisencriterion => not eisensteinIrreducible?(g) ;
                true)

            then

                -- g may be reducible

                llg := HenselLift(g,lg,cprime,b)

                gpk: P := (llg.modulo)::P

                -- In case exact factorisation has been reached by

                -- HenselLift before coefficient bound.

                if gpk<b then

                    lg := llg.plist

                else

                    lg := completeFactor(g,llg.plist,cprime,gpk,rg,dg)

                else

                    lg := [ g ] -- g irreducible

            end if

        else lg := [ g ] -- g irreducible

        end if

        ltrue := append(ltrue,lg)

        r := max(2,(r-#lg))::N

        degef := degree f

        d := select(x+->x <= degef,d)

        if degef<=1 then -- if exhausted

            if one? degef then

                if (degef = 1) then
ltrue := cons(f,ltrue)
return ltrue -- 1st exit, all factors found
else -- can we go on with the same pk?
b := (2*lc*singleFactorBound(f,r)) :: P
if b>pk then -- unlucky: no we can't
lg := HenselLift(f,lf,cprime,b) -- I should reformulate
-- the lifting problem, but hadn't time for that.
-- In any case, such case should be quite exceptional.
lf := lg.plist
pk := (lg.modulo)::P
-- In case exact factorisation has been reached by
-- HenselLift before coefficient bound.
if pk<b then return append(lf,ltrue) -- 2nd exit
level := 1
ic := nextRecNum(levels,level,i)
if found? then
levels := #lf
found? := false
if not (ic case Z) then level := level+1
cons(f,ltrue) -- 3rd exit, the last factor was irreducible but not "true"

-- local function, returns the set of elements "divided" by an integer
divideSet(s:Set N, n:N):Set N ==
l: List N := [0]
for e in parts s repeat
if (ee := (e exquo n)$N) case N then l := cons(ee::N,l)
set(l)

-- Beauzamy-Trevisan-Wang FACTOR, see [1] with some refinements
-- and some differences. f is assumed to be primitive, square-free
-- and with positive leading coefficient. If pdecomp is true then
-- look for a functional decomposition of f.
btwFactor(f:UP,d:Set N,r:N,pdecomp:Boolean):List UP ==
if degree f
not (max(d) = df) => error "btwFact: Bad arguments"
reverse?: Boolean := false
negativelc?: Boolean := false

df := degree f
if abs(coefficient(f,0))<abs(leadingCoefficient(f)) then
f := reverse f
reverse? := true
brillhartIrreducible? f or
(useeisensteincritereon => eisensteinIrreducible?(f) ; false) =>
if reverse? then [reverse f] else [f]
if leadingCoefficient(f)<0 then
f := -f
negativelc? := true
cf: Union(LR,"failed")
if pdecomp and tryfunctionaldecomposition then
cf := monicDecomposeIfCan f
else
cf := "failed"
if cf case "failed" then
m := modularFactor f
zero? (cprime := m.prime) =>
if reverse? then
if negative|lc? then return [ -reverse f ]
else return [ reverse f ]
else if negative|lc? then return [ -f ]
else return [ f ]
if noLinearFactor? f then d := remove(1,d)
lc := leadingCoefficient f
f0 := coefficient(f,0)
b: P := (2*lc*singleFactorBound(f,r)) :: P -- LC algorithm
lm := HenselLift(f,m.factors,cprime,b)
lf := lm.plist
pk: P := (lm.modulo)::P
if ground? first lf then lf := rest lf
-- in case exact factorisation has been reached by HenselLift
-- before coefficient bound
if not pk < b then lf := completeFactor(f,lf,cprime,pk,r,d)
else
lrf := cf::LR
dh := degree lrf.right
lg := btwFactor(lrf.left,divideSet(d,dh),2,true)
lf: List UP := empty()
for i in 1..#lg repeat
g := lg.i
dgh := (degree g)*dh
df := subtractIfCan(df,dgh)::N
lfg := btwFactor(g(lrf.right),
select(x+->x <= dgh,d),max(2,r-df)::N,false)
lf := append(lf,lfg)
r := max(2,r-#lfg)::N
if reverse? then lf := [ reverse(fact) for fact in lf ]
for i in 1..#lf repeat
if leadingCoefficient(lf.i)<0 then lf.i := -lf.i
-- because we assume f with positive leading coefficient
lf
makeFR(flist:FinalFact):Factored UP ==
ctp := factor flist.contp
fflist: List FFE := empty()
for ff in flist.factors repeat
fflist := cons(["prime", ff.irr, ff.pow]::$FFE, ffflist)
for fc in factorList ctp repeat
fflist := cons([fc.fig, fc.fctr::UP, fc.xpnt]::$FFE, ffflist)
makeFR(unit(ctp)::UP, ffflist)
import IntegerRoots(Z)

-- local function, factorizes a quadratic polynomial
quadratic(p:UP):List UP ==
a := leadingCoefficient p
b := coefficient(p,1)
d := b**2-4*a*coefficient(p,0)
r := perfectSqrt(d)
r case "failed" => [p]
b := b+(r::Z)
a := 2*a
d := gcd(a,b)
-- if not one? d then
if not (d = 1) then
  a := a quo d
  b := b quo d
f := UP := monomial(a,1)+monomial(b,0)
cons(f,[(p exquo f)::UP])

isPowerOf2(n:Z): Boolean ==
n = 1 => true
qr: Record(quotient: Z, remainder: Z) := divide(n,2)
qr.remainder = 1 => false
isPowerOf2 qr.quotient

subMinusX(supPol: SUPZ): UP ==
  minusX: SUPZ := monomial(-1,1)$SUPZ
  unmakeSUP(elt(supPol,minusX)$SUPZ)

henselFact(f:UP, sqf:Boolean):FinalFact ==
factorlist: List(ParFact) := empty()
-- make m primitive
c := content f
f := (f exquo c)::UP

-- make the leading coefficient positive
if leadingCoefficient f < 0 then
  c := -c
  f := -f

-- is x**d factor of f
if (d := minimumDegree f) > 0 then
  f := monicDivide(f,monomial(1,d)).quotient
  factorlist := [[monomial(1,1),d]$ParFact]
d := degree f

-- is f constant?
zero? d => [c,factorlist]$FinalFact
-- is f linear?
-- one? d ⇒ [c,cons([[f,1]$ParFact,factorlist]]$FinalFact
(d = 1) ⇒ [c,cons([[f,1]$ParFact,factorlist]]$FinalFact

lcPol: UP := leadingCoefficient(f) :: UP

-- is f cyclotomic (x^n - 1)?
-lcPol = reductum(f) ⇒ -- if true, both will = 1
for fac in map(z+->unmakeSUP(z)$UP,
cyclotomicDecomposition(d)$CYC)$ListFunctions2(SUPZ,UP) repeat
factorlist := cons([[fac,1]$ParFact,factorlist]]$FinalFact
[c,factorlist]$FinalFact

-- is f odd cyclotomic (x**(2*n+1) + 1)?
odd?(d) and (lcPol = reductum(f)) ⇒
for sfac in cyclotomicDecomposition(d)$CYC repeat
fac := subMinusX sfac
if leadingCoefficient fac < 0 then fac := -fac
factorlist := cons([[fac,1]$ParFact,factorlist]]$FinalFact
[c,factorlist]$FinalFact

-- is the poly of the form x**n + 1 with n a power of 2?
-- if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(f)) ⇒
factorlist := cons([[f,1]$ParFact,factorlist]]$FinalFact

-- other special cases to implement...

-- f is square-free :
sqf ⇒ [c, append([[pf,1]$ParFact for pf in henselfact(f,true)],
factorlist)]$FinalFact

-- f is not square-free :
sqfflist := factors squareFree f
for sqfr in sqfflist repeat
mult := sqfr.exponent
sqff := sqfr.factor
d := degree sqff
-- one? d ⇒ factorlist := cons([[sqff,mult]$ParFact,factorlist]]$FinalFact
(d = 1) ⇒ factorlist := cons([[sqff,mult]$ParFact,factorlist]]$FinalFact
(d=2 ⇒
factorlist := append([[pf,mult]$ParFact for pf in quadratic(sqff)],
factorlist)
factorlist := append([[pf,mult]$ParFact for pf in henselfact(sqff,true)],factorlist)
[c,factorlist]$FinalFact

btwFact(f:UP, sqf:Boolean, fd:Set N, r:N):FinalFact ==
d := degree f
not(max(fd)=d) => error "btwFact: Bad arguments"
factorlist: List(ParFact) := empty()

-- make m primitive
c: Z := content f
f := (f exquo c)::UP

-- make the leading coefficient positive
if leadingCoefficient f < 0 then
  c := -c
  f := -f

-- is x**d factor of f
if (maxd := minimumDegree f) > 0 then
  f := monicDivide(f,monomial(1,maxd)).quotient
  factorlist := [[monomial(1,1),maxd]$ParFact]
  r := max(2,r-maxd)::N
  d := subtractIfCan(d,maxd)::N
  fd := select(x+->x <= d,fd)

-- is f constant?
zero? d => [c,factorlist]$FinalFact

-- is f linear?
one? d => [c,cons([f,1]$ParFact,factorlist)]$FinalFact
(d = 1) => [c,cons([f,1]$ParFact,factorlist)]$FinalFact

lcPol: UP := leadingCoefficient(f) :: UP

-- is f cyclotomic (x**n - 1)?
-lcPol = reductum(f) =>
  for fac in cyclotomicDecomposition(d)$CYC repeat
    factorlist := cons([[fac,1]$ParFact,factorlist])
  [c,factorlist]$FinalFact

-- is f odd cyclotomic (x**(2*n+1) + 1)?
odd?(d) and (lcPol = reductum(f)) =>
  for sfac in cyclotomicDecomposition(d)$CYC repeat
    fac := subMinusX sfac
    if leadingCoefficient fac < 0 then fac := -fac
    factorlist := cons([[fac,1]$ParFact,factorlist])
  [c,factorlist]$FinalFact

-- is the poly of the form x**n + 1 with n a power of 2?
-- if so, then irreducible
isPower0f2(d) and (lcPol = reductum(f)) =>
  factorlist := cons([[f,1]$ParFact,factorlist])
  [c,factorlist]$FinalFact
-- other special cases to implement...

-- f is square-free:
  sqf => [c, append([[[pf,1]$ParFact for pf in btwFactor(f,fd,r,true)],
                  factorlist])$FinalFact

-- f is not square-free:
  sqfflist := factors squareFree(f)
  if one?(#(sqfflist)) then -- indeed f was a power of a square-free
    r := max(r quo ((first sqfflist).exponent),2)::N
  else
    r := 2
  for sqfr in sqfflist repeat
    mult := sqfr.exponent
    sqff := sqfr.factor
    d := degree sqff
    if (d = 1) =>
      factorlist := cons([[sqff,mult]$ParFact,factorlist)
     maxd := (max(fd)-mult)::N
     fd := select(x+->x <= maxd,fd)
    d=2 =>
      factorlist := append([[[pf,mult]$ParFact for pf in quadratic(sqff)],
                         factorlist)
     maxd := (max(fd)-2*mult)::N
     fd := select(x+->x <= maxd,fd)
    factorlist := append([[[pf,mult]$ParFact for pf in 
                           btwFactor(sqff,select(x+->x <= d,fd),r,true)],factorlist)
     maxd := (max(fd)-d*mult)::N
     fd := select(x+->x <= maxd,fd)
  [c,factorlist]$FinalFact

factor(f:UP):Factored UP ==
  makeFR
    usesinglefactorbound => btwFact(f,false,fullSet(degree f),2)
    henselFact(f,false)

-- local function, returns true if the sum of the elements of the list
-- is not the degree.
errorsum?(d:N,ld:List N):Boolean == not (d = +/ld)

-- local function, turns list of degrees into a Set
makeSet(ld:List N):Set N ==
  s := set [0]
  for d in ld repeat s := union(s,shiftSet(s,d))
  s

factor(f:UP,ld:List N,r:N):Factored UP ==
factor(f:UP,r:N):Factored UP == makeFR btwFact(f,false,fullSet(degree f),r)

factor(f:UP,ld:List N):Factored UP == factor(f,ld,2)

factor(f:UP,d:N,r:N):Factored UP ==
    n := (degree f) exquo d
    n case "failed" => error "factor: Bad arguments"
    factor(f,new(n::N,d)$List(N),r)

factorSquareFree(f:UP):Factored UP ==
    makeFR
    usesinglefactorbound => btwFact(f,true,fullSet(degree f),2)
    henselFact(f,true)

factorSquareFree(f:UP,ld:List(N),r:N):Factored UP ==
    errorsum?(degree f,ld) => error "factorSquareFree: Bad arguments"
    makeFR btwFact(f,true,makeSet(ld),r)

factorSquareFree(f:UP,r:N):Factored UP ==
    makeFR btwFact(f,true,fullSet(degree f),r)

factorSquareFree(f:UP,ld:List N):Factored UP == factorSquareFree(f,ld,2)

factorSquareFree(f:UP,d:N,r:N):Factored UP ==
    n := (degree f) exquo d
    n case "failed" => error "factorSquareFree: Bad arguments"
    factorSquareFree(f,new(n::N,d)$List(N),r)

    dp := degree p
    errorsum?(dp,ld) => error "factorOfDegree: Bad arguments"
    (one? (d::N)) and noLinearFactor?(p) => "failed"
    ((d::N) = 1) and noLinearFactor?(p) => "failed"
    if := btwFact(p,sqf,makeSet(ld),r).factors
    for f in if repeat
        degree(f.irr)=d => return f.irr
    "failed"

    factorOfDegree(d,p,ld,r,false)

factorOfDegree(d:P,p:UP,r:N):Union(UP,"failed") ==
    factorOfDegree(d,p,new(degree p,1)$List(N),r,false)

    factorOfDegree(d,p,ld,2,false)
factorOfDegree(d:p,p:UP):Union(UP,"failed") ==
factorOfDegree(d,p,new(degree p,1)$List(N),2,false)

---

— GALFACT.dotabb —

"GALFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALFACT"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"GALFACT" -> "FSAGG"

---

package GALPOLYU GaloisGroupPolynomialUtilities

— GaloisGroupPolynomialUtilities.input —

)set break resume
)sys rm -f GaloisGroupPolynomialUtilities.output
)spool GaloisGroupPolynomialUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupPolynomialUtilities
--E 1

)spool
)lisp (bye)

---

— GaloisGroupPolynomialUtilities.help —

====================================================================
GaloisGroupPolynomialUtilities examples
====================================================================

GaloisGroupPolynomialUtilities provides useful functions for
univariate polynomials which should be added to
UnivariatePolynomialCategory or to Factored.
See Also:
- )show GaloisGroupPolynomialUtilities

---

GaloisGroupPolynomialUtilities (GALPOLYU)

Exports:
- degreePartition
- factorOfDegree
- factorsOfDegree
- monic?
- reverse
- scaleRoots
- shiftRoots
- unvectorise

--- package GALPOLYU GaloisGroupPolynomialUtilities ---

)abbrev package GALPOLYU GaloisGroupPolynomialUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 15 July 1994
++ Description:
++ \spadtype{GaloisGroupPolynomialUtilities} provides useful
++ functions for univariate polynomials which should be added to
++ \spadtype{UnivariatePolynomialCategory} or to \spadtype{Factored}

GaloisGroupPolynomialUtilities(R,UP): Exports == Implementation where
R : Ring
UP : UnivariatePolynomialCategory R
N ==> NonNegativeInteger
P ==> PositiveInteger

Exports => with
- monic?: UP -> Boolean
  ++ monic?(p) tests if p is monic (i.e. leading coefficient equal to 1).
- unvectorise: Vector R -> UP
  ++ unvectorise(v) returns the polynomial which has for coefficients the
  ++ entries of v in the increasing order.
reverse: UP -> UP
++ reverse(p) returns the reverse polynomial of p.
scaleRoots: (UP,R) -> UP
++ scaleRoots(p,c) returns the polynomial which has c times the roots
++ of p.
shiftRoots: (UP,R) -> UP
++ shiftRoots(p,c) returns the polynomial which has for roots c added
++ to the roots of p.
degreePartition: Factored UP -> Multiset N
++ degreePartition(f) returns the degree partition (i.e. the multiset
++ of the degrees of the irreducible factors) of
++ the polynomial f.
factorOfDegree: (P, Factored UP) -> UP
++ factorOfDegree(d,f) returns a factor of degree d of the factored
++ polynomial f. Such a factor shall exist.
factorsOfDegree: (P, Factored UP) -> List UP
++ factorsOfDegree(d,f) returns the factors of degree d of the factored
++ polynomial f.

Implementation ==> add

import Factored UP

factorsOfDegree(d:P,r:Factored UP):List UP ==
lfact : List UP := empty()
for fr in factors r | degree(fr.factor)=(d::N) repeat
  for i in 1..fr.exponent repeat
    lfact := cons(fr.factor,lfact)
lfact

factorOfDegree(d:P,r:Factored UP):UP ==
factor : UP := 0
for i in 1..numberOfFactors r repeat
  factor := nthFactor(r,i)
  if degree(factor)=(d::N) then return factor
error "factorOfDegree: Bad arguments"

degreePartition(r:Factored UP):Multiset N ==
multiset([ degree(nthFactor(r,i)) for i in 1..numberOfFactors r ])

-- monic?(p:UP):Boolean == one? leadingCoefficient p
monic?(p:UP):Boolean == (leadingCoefficient p) = 1

unvectorise(v:Vector R):UP ==
p : UP := 0
for i in 1..#v repeat p := p + monomial(v(i),(i-1)::N)
p

reverse(p:UP):UP ==
r : UP := 0
n := degree(p)
for i in 0..n repeat r := r + monomial(coefficient(p,(n-i)::N),i)
r
scaleRoots(p:UP,c:R):UP ==
   -- one? c => p
   (c = 1) => p
   n := degree p
   zero? c => monomial(leadingCoefficient p,n)
r : UP := 0
mc : R := 1
for i in n..0 by -1 repeat
   r := r + monomial(mc*coefficient(p,i),i)
   mc := mc*c
r

import UnivariatePolynomialCategoryFunctions2(R,UP,UP,
   SparseUnivariatePolynomial UP)

shiftRoots(p:UP,c:R):UP == elt(map(coerce,p),monomial(1,1)$UP-c::UP)::UP

package GALUTIL GaloisGroupUtilities

— GaloisGroupUtilities.input —

)set break resume
)sys rm -f GaloisGroupUtilities.output
)spool GaloisGroupUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupUtilities
GaloisGroupUtilities (GALUTIL)

Exports:
- fillPascalTriangle
- pascalTriangle
- rangePascalTriangle
- safeCeiling
- safeFloor
- safetyMargin
- sizePascalTriangle

---

--E 1
)spool
)lisp (bye)

---

---

---

---

---
\texttt{GaloisGroupUtilities} provides several useful functions.

\texttt{GaloisGroupUtilities(R): Exports == Implementation where}
\begin{itemize}
  \item \texttt{N \rightarrow NonNegativeInteger}
  \item \texttt{Z \rightarrow Integer}
  \item \texttt{R : Ring}
\end{itemize}

\begin{itemize}
  \item \texttt{Exports \rightarrow with}
    \begin{itemize}
      \item \texttt{pascalTriangle: (N,Z) \rightarrow R}
        \begin{itemize}
          \item \texttt{pascalTriangle(n,r) returns the binomial coefficient}
          \item \texttt{C(n,r)=n!/r!(n-r)!}
          \item \texttt{and stores it in a table to prevent recomputation.}
        \end{itemize}
      \item \texttt{rangePascalTriangle: N \rightarrow N}
        \begin{itemize}
          \item \texttt{rangePascalTriangle(n) sets the maximal number of lines which}
          \item \texttt{are stored and returns the previous value.}
        \end{itemize}
      \item \texttt{rangePascalTriangle: () \rightarrow N}
        \begin{itemize}
          \item \texttt{rangePascalTriangle() returns the maximal number of lines stored.}
        \end{itemize}
      \item \texttt{sizePascalTriangle: () \rightarrow N}
        \begin{itemize}
          \item \texttt{sizePascalTriangle() returns the number of entries currently stored}
          \item \texttt{in the table.}
        \end{itemize}
      \item \texttt{fillPascalTriangle: () \rightarrow Void}
        \begin{itemize}
          \item \texttt{fillPascalTriangle() fills the stored table.}
        \end{itemize}
    \end{itemize}
  \end{itemize}

\texttt{Implementation \rightarrow add}

\begin{itemize}
  \item \texttt{if R has FloatingPointSystem then}
    \begin{itemize}
      \item \texttt{safeCeiling: R \rightarrow Z}
        \begin{itemize}
          \item \texttt{safeCeiling(x) returns the integer which is greater than any integer}
          \item \texttt{with the same floating point number representation.}
        \end{itemize}
      \item \texttt{safeFloor: R \rightarrow Z}
        \begin{itemize}
          \item \texttt{safeFloor(x) returns the integer which is lower or equal to the}
          \item \texttt{largest integer which has the same floating point number}
          \item \texttt{representation.}
        \end{itemize}
      \item \texttt{safetyMargin: N \rightarrow N}
        \begin{itemize}
          \item \texttt{safetyMargin(n) sets to n the number of low weight digits we do not}
          \item \texttt{trust in the floating point representation and returns the previous}
          \item \texttt{value (for use by \texttt{safeCeiling}).}
        \end{itemize}
    \end{itemize}
  \end{itemize}

\texttt{Implementation \rightarrow add}

\begin{itemize}
  \item \texttt{if R has FloatingPointSystem then}
    \begin{itemize}
      \item \texttt{safetymargin \rightarrow N := 6}
    \end{itemize}
  \end{itemize}

\texttt{safeFloor(x:R):Z ==}
\begin{itemize}
  \item \texttt{if (shift := order(x)-precision()+safetymargin) >= 0 then}
    \begin{itemize}
      \item \texttt{x := x+float(1,shift)}
      \item \texttt{retract(floor(x))@Z}
    \end{itemize}
\end{itemize}
safeCeiling(x:R):Z ==
  if (shift := order(x)-precision()$R+safetymargin) >= 0 then
    x := x+float(1,shift)
  retract(ceiling(x))@Z

safetyMargin(n:N):N ==
  (safetymargin,n) := (n,safetymargin)
  n

safetyMargin():N == safetymargin

pascaltriangle : FlexibleArray(R) := empty()
ncomputed : N := 3
rangepascaltriangle : N := 216

pascalTriangle(n:N, r:Z):R ==
  negative? r => 0
  (d := n-r) < r => pascalTriangle(n,d)
  zero? r => 1$R
  -- one? r => n :: R
  (r = 1) => n :: R
  n > rangepascaltriangle =>
    binomial(n,r)$IntegerCombinatoricFunctions(Z) :: R
  n <= ncomputed =>
    m := divide(n-4,2)
    mq := m.quotient
    pascaltriangle((mq+1)*(mq+m.remainder)+r-1)
  -- compute the missing lines
  for i in (ncomputed+1)..n repeat
    for j in 2..(i quo 2) repeat
      pascaltriangle := concat!(pascaltriangle,pascalTriangle((i-1) :: N, j-1)+pascalTriangle((i-1) :: N,j))
    ncomputed := i
  pascalTriangle(n,r)

rangePascalTriangle(n:N):N ==
  if n<ncomputed then
    if n<3 then
      pascaltriangle := delete!(pascaltriangle,1..#pascaltriangle)
      ncomputed := 3
    else
      d := divide(n-3,2)
      dq := d.quotient
      pascaltriangle := delete!(pascaltriangle,((dq+1) *(dq+d.remainder) +1)..#pascaltriangle)
      ncomputed := n
    (rangepascaltriangle,n) := (n,rangepascaltriangle)
  n

rangePascalTriangle():N == rangepascaltriangle
sizePascalTriangle():N == #pascaltriangle

fillPascalTriangle():Void == pascalTriangle(rangepascaltriangle,2)

| \--- GALUTIL.dotabb \--- |
| "GALUTIL" \[color=FF4488,href=bookvol10.4.pdf#nameddest=GALUTIL\] |
| "A1AGG" \[color=#4488FF,href=bookvol10.2.pdf#nameddest=A1AGG\] |
| "GALUTIL" \rightarrow "A1AGG" |

--S 1 of 1
)show GaussianFactorizationPackage
--E 1

--- GaussianFactorizationPackage.help ---

GaussianFactorizationPackage examples

Package for the factorization of complex or gaussian integers.
See Also:
  o )show GaussianFactorizationPackage

GaussianFactorizationPackage (GAUSSFAC)

Exports:
  factor  prime?  sumSquares

--- package GAUSSFAC GaussianFactorizationPackage ---

)abbrev package GAUSSFAC GaussianFactorizationPackage
++ Author: Patrizia Gianni
++ Date Created: Summer 1986
++ Description:
++ Package for the factorization of complex or gaussian integers.

GaussianFactorizationPackage(): C == T
where
  NNI  ==  NonNegativeInteger
  Z    ==>  Integer
  ZI   ==>  Complex Z
  FRZ  ==>  Factored ZI
  fUnion == Union("nil", "sqfr", "irred", "prime")
  FFE   ==>  Record(flg:fUnion, fctr:ZI, xpnt:Integer)

C == with
  factor : ZI -> FRZ
    ++ factor(zi) produces the complete factorization of the complex
    ++ integer zi.
  sumSquares : Z -> List Z
    ++ sumSquares(p) construct \spad{a} and b such that \spad{a**2+b**2}
prime? : ZI -> Boolean
++ prime?(zi) tests if the complex integer zi is prime.

T == add
import IntegerFactorizationPackage Z

reduction(u:Z,p:Z):Z ==
p=0 => u
positiveRemainder(u,p)

merge(p:Z,q:Z):Union(Z,"failed") ==
p = q => p
p = 0 => q
q = 0 => p
"failed"

p=0 => u exquo v
v rem p = 0 => "failed"
positiveRemainder((extendedEuclidean(v,p,u)::Record(coef1:Z,coef2:Z)).coef1,p)

FMod := ModularRing(Z,Z,reduction,merge,exactquo)

fact2:ZI:= complex(1,1)

---- find the solution of x**2+1 mod q ----
findelt(q:Z) : Z ==
q1:=q-1
r:=q1
r1:=r exquo 4
while ~(r1 case "failed") repeat
  r:=r1::Z
  r1:=r exquo 2
s : FMod := reduce(1,q)
qq1:FMod :=reduce(q1,q)
for i in 2.. while (s=1 or s=qq1) repeat
  s:=reduce(i,q)**(r::NNI)
t:=s
while t^=qq1 repeat
  s:=t
t:=t**2
s::Z

---- write p, congruent to 1 mod 4, as a sum of two squares ----
sumsq1(p:Z) : List Z ==
s:= findelt(p)  
u:=p  
while u**2>p repeat  
w:=u rem s  
u:=s  
s:=w  
[u,s]  

---- factorization of an integer ----  
intfactor(n:Z) : Factored ZI ==  
  lfn:= factor n  
  r : List FFE :=[]  
  unity:ZI:=complex(unit lfn,0)  
  for term in (factorList lfn) repeat  
    n:=term.fctr  
    exp:=term.xpnt  
    n=2 =>  
      r :=concat(["prime",fact2,2*exp]$FFE,r)  
      unity:=unity*complex(0,-1)**(exp rem 4)::NNI  
    (n rem 4) = 3 => r:=concat(["prime",complex(n,0),exp]$FFE,r)  
    sz:=sumsq1(n)  
    z:=complex(sz.1,sz.2)  
    r:=concat(["prime",z,exp]$FFE,  
                concat(["prime",conjugate(z),exp]$FFE,r))  
  makeFR(unity,r)  

---- factorization of a gaussian number ----  
factor(m:ZI) : FRZ ==  
  m=0 => primeFactor(0,1)  
  a:= real m  
  (b:= imag m)=0 => intfactor(a) :: FRZ  
  a=0 =>  
    ris:=intfactor(b)  
    unity:= unit(ris)*complex(0,1)  
    makeFR(unity,factorList ris)  
  d:=gcd(a,b)  
  result : List FFE :=[]  
  unity:ZI:=1$ZI  
  if d"=1 then  
    a:=(a exquo d)::Z  
    b:=(b exquo d)::Z  
    r:= intfactor(d)  
    result:=factorList r  
    unity:=unit r
m := complex(a,b)

n := a**2 + b**2
factn := factorList(factor n)
part := ["prime", 0$ZI, 0]
for term in factn repeat
  n := term.fctr
  exp := term.xpnt
  n = 2 =>
    part := ["prime", fact2, exp]$FFE
    m := m quo (fact2**exp:NNI)
    result := concat(part, result)
(n rem 4) = 3 =>
  g0 := complex(n, 0)
  part := ["prime", g0, exp quo 2]$FFE
  m := m quo g0
  result := concat(part, result)

z := gcd(m, complex(n, 0))
part := ["prime", z, exp]$FFE
z := z**(exp:NNI)
result := concat(part, result)
if m^=1 then unity := unity * m
makeFR(unity, result)

---- write p prime like sum of two squares ----
sumSquares(p:Z) : List Z ==
p := 2 => [1,1]
p rem 4 ^= 1 => error "no solutions"
sumsq1(p)

prime?(a:ZI) : Boolean ==
n := norm a
n = 0 => false -- zero
n = 1 => false -- units
prime?(n)$IntegerPrimesPackage(Z) => true
re := real a
im := imag a
re ^= 0 and im ^= 0 => false
p := abs(re+im) -- a is of the form p, -p, %i*p or -%i*p
p rem 4 ^= 3 => false
-- return-value true, if p is a rational prime,
-- and false, otherwise
prime?(p)$IntegerPrimesPackage(Z)
package GHENSEL GeneralHenselPackage

---

GeneralHenselPackage examples

Used for Factorization of bivariate polynomials over a finite field.

See Also:
  o )show GeneralHenselPackage
GeneralHenselPackage (GHENSEL)

Exports:
  completeHensel  HenselLift  reduction

— package GHENSEL GeneralHenselPackage —

)abbrev package GHENSEL GeneralHenselPackage
++ Author : P.Gianni
++ Description:
  ++ General Hensel Lifting
  ++ Used for Factorization of bivariate polynomials over a finite field.

GeneralHenselPackage(RP,TP):C == T where
  RP : EuclideanDomain
  TP : UnivariatePolynomialCategory RP

  PI == PositiveInteger

  C == with
    HenselLift: (TP,List(TP),RP,PI) -> Record(plist:List(TP), modulo:RP)
      ++ HenselLift(pol,lfacts,prime,bound) lifts lfacts, 
      ++ that are the factors of pol mod prime,
      ++ to factors of pol mod prime**k > bound. No recombining is done.

    completeHensel: (TP,List(TP),RP,PI) -> List TP
      ++ completeHensel(pol,lfact,prime,bound) lifts lfact,
      ++ the factorization mod prime of pol,
      ++ to the factorization mod prime**k>bound.
      ++ Factors are recombined on the way.

    reduction : (TP,RP) -> TP
      ++ reduction(u,pol) computes the symmetric reduction of u mod pol

  T == add
    GenExEuclid: (List(FP),List(FP),FP) -> List(FP)
    HenselLift1: (TP,List(TP),List(FP),List(FP),RP,RP,F) -> List(TP)
mQuo: (TP,RP) -> TP

reduceCoeff(c:RP,p:RP):RP ==
  zero? p => c
  RP is Integer => symmetricRemainder(c,p)
  c rem p

reduction(u:TP,p:RP):TP ==
  zero? p => u
  RP is Integer => map(x+->symmetricRemainder(x,p),u)
  map(x+->x rem p,u)

merge(p:RP,q:RP):Union(RP,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

modInverse(c:RP,p:RP):RP ==
  (extendedEuclidean(c,p,1)::Record(coef1:RP,coef2:RP)).coef1

exactquo(u:TP,v:TP,p:RP):Union(TP,"failed") ==
  invlcv:=modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlcv*v,p))
  reduction(r.remainder,p) ^=0 => "failed"
  reduction(invlcv*r.quotient,p)

FP:=EuclideanModularRing(RP,TP,RP,reduction,merge,exactquo)
mQuo(poly:TP,n:RP) : TP == map(x+->x quo n,poly)

GenExEuclid(fl:List FP,cl:List FP,rhs:FP) :List FP ==
  [clp*rhs rem flp for clp in cl for flp in fl]

-- generate the possible factors
genFact(fln:List TP,factlist:List List TP) : List List TP ==
  factlist=[] => [[pol] for pol in fln]
  maxd := +/[degree f for f in fln] quo 2
  auxfl:List List TP := []
  for poly in fln while factlist=[] repeat
    factlist := [term for term in factlist | ^member?(poly,term)]
    dp := degree poly
    for term in factlist repeat
      (+/[degree f for f in term]) + dp > maxd => "next term"
      auxfl := cons(cons(poly,term),auxfl)
  auxfl

HenselLift1(poly:TP,fln:List TP,fl1:List FP,cl1:List FP,
  lcp := leadingCoefficient poly
rhs := reduce(mQuo(poly - lcp */ fln, Modulus), prime)
zero? rhs => fln
lciinv := reduce(cinv::TP, prime)
vl := GenExEuclid(fl1, cl1, lciinv*rhs)
[flp + Modulus*(vlp::TP) for flp in fln for vlp in vl]

HenselLift(poly: TP, tl1: List TP, prime: RP, bound: PI) ==
  -- convert tl1
  constp: TP := 0
  if degree first tl1 = 0 then
    constp := tl1.first
    tl1 := rest tl1
  fl1 := [reduce(ttl, prime) for ttl in tl1]
  cl1 := multiEuclidean(fl1, 1):: List FP
  Modulus := prime
  fln := List TP := [ffl1::TP for ffl1 in fl1]
  lciinv := retract((inv (reduce((leadingCoefficient poly)::TP, prime)))::TP)
  while euclideanSize(Modulus) < bound repeat
    nfln := HenselLift1(poly, fln, fl1, cl1, prime, Modulus, lciinv)
    fln = nfln and zero?(err := poly - */ fln) => leave "finished"
    fln := nfln
    Modulus := prime*Modulus
  if constp ^= 0 then fln := cons(constp, fln)
  [fln, Modulus]

completeHensel(m: TP, tl1: List TP, prime: RP, bound: PI) ==
  hlift := HenselLift(m, tl1, prime, bound)
  Modulus := hlift.modulo
  fln := List TP := hlift.plist
  nm := degree m
  u := Union(TP, "failed")
  aux, auxl, finallist := List TP
  auxfl, factlist := List List TP
  factlist := []
  dfn := NonNegativeInteger := nm
  lci1 := leadingCoefficient m
  mm := lci1*m
  while dfn > 0 and (factlist := genFact(fl1, factlist)) ^= [] repeat
    auxfl := []
    while factlist ^= [] repeat
      auxl := factlist.first
      factlist := factlist.rest
      tc := reduceCoef((lci1 * */ [coefficient(poly, 0)
                                 for poly in auxl]), Modulus)
      coefficient(mm, 0) exquo tc case "failed" =>
        auxfl := cons(auxl, auxfl)
        pol := */ [poly for poly in auxl]
        poly := reduction(lci1*pol, Modulus)
        u := mm exquo poly
u case "failed" => auxl := cons(aux1,auxl)
poly1: TP := primitivePart poly
m := mQuo((u::TP),leadingCoefficient poly1)
lcm1 := leadingCoefficient(m)
mm := lcm1*m
finallist := cons(poly1,finallist)
dfn := degree m
aux := []
for poly in fln repeat
  ^member?(poly,auxl) => aux := cons(poly,aux)
auxl := [term for term in auxfl | ^member?(poly,term)]
factlist := [term for term in factlist | ^member?(poly,term)]
fln := aux
factlist := auxfl
if dfn > 0 then finallist := cons(m,finallist)
finallist

— GHENSEL.dotabb —

"GHENSEL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GHENSEL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GHENSEL" -> "PFECAT"

— GeneralizedMultivariateFactorize.input —

)set break resume
)sys rm -f GeneralizedMultivariateFactorize.output
)spool GeneralizedMultivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralizedMultivariateFactorize
--E 1

)spool
)lisp (bye)
GeneralizedMultivariateFactorize (GENMFACT)

Exports:
factor

--- package GENMFACT GeneralizedMultivariateFactorize ---

)abbrev package GENMFACT GeneralizedMultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \spadtype{Integer} or \spadtype{Fraction Integer}.

GeneralizedMultivariateFactorize(OV,E,S,R,P) : C == T
where
package GPAFF GeneralPackageForAlgebraicFunctionField

--- GeneralPackageForAlgebraicFunctionField.input ---

)set break resume
CHAPTER 8.  CHAPTER G

)sys rm -f GeneralPackageForAlgebraicFunctionField.output
)spool GeneralPackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all

--)S 1 of 1
)show GeneralPackageForAlgebraicFunctionField
--)R
--)R GeneralPackageForAlgebraicFunctionField(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory) is GPAFF
--)R This constructor is exposed in this frame.
--)R Issue )edit bookvol10.4.pamphlet to see algebra source code for GPAFF
--)R

--R------------------------------------------------------------- Operations --------------------------------
--R adjunctionDivisor : () -> DIVISOR desingTree : () -> List(DesTree)
--R eval : (PolyRing,Plc) -> K genus : () -> NonNegativeInteger
--R genusNeg : () -> Integer placesAbove : ProjPt -> List(Plc)
--R pointDominateBy : Plc -> ProjPt printInfo : List(Boolean) -> Void
--R reset : () -> Void setCurve : PolyRing -> PolyRing
--R theCurve : () -> PolyRing
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R classNumber : () -> Integer if K has FINITE
--R desingTreeWoFullParam : () -> List(DesTree)
--R eval : (PolyRing,PolyRing,Plc) -> K
--R eval : (Fraction(PolyRing),Plc) -> K
--R evaluCan : (PolyRing,Plc) -> Union(K,"failed")
--R evaluCan : (PolyRing,PolyRing,Plc) -> Union(K,"failed")
--R evaluCan : (Fraction(PolyRing),Plc) -> Union(K,"failed")
--R findOrderOfDivisor : (DIVISOR,Integer,Integer) -> Record(ord: Integer,num: PolyRing,den: PolyRing)
--R homogenize : (PolyRing,Integer) -> PolyRing
--R interpolateForms : (DIVISOR,NonNegativeInteger) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R intersectionDivisor : PolyRing -> DIVISOR
--R lBasis : DIVISOR -> List(PolyRing)
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R parametrize : (PolyRing,Plc) -> PCS
--R placesOfDegree : PositiveInteger -> List(Plc) if K has FINITE
--R rationalPlaces : () -> List(Plc) if K has FINITE
--R rationalPoints : () -> List(ProjPt) if K has FINITE
--R setSingularPoints : List(ProjPt) -> List(ProjPt)
--R singularPoints : () -> List(ProjPt)
--R
--E 1
A package that implements the Brill-Noether algorithm.
Part of the PAFF package.

See Also:
- )show GeneralPackageForAlgebraicFunctionField

---

**GeneralPackageForAlgebraicFunctionField (GPAFF)**

- adjunctionDivisor
- desingTreeWoFullParam
- findOrderOfDivisor
- homogenize
- intersectionDivisor
- numberOfPlacesOfDegree
- parametrize
- pointDominateBy
- rationalPoints
- setSingularPoints
- ZetaFunction
- classNumber
- eval
- genus
- interpolateForms
- lBasis
- numberPlacesDegExtDeg
- placesAbove
- printInfo
- reset
- singularPoints
- desingTree
- evalIfCan
- genusNeg
- interpolateFormsForFact
- LPolynomial
- numberRatPlacesExtDeg
- placesOfDegree
- rationalPlaces
- setCurve
- theCurve
Abbrev package GPAFF GeneralPackageForAlgebraicFunctionField

Author: Gaetan Hache
Date created: June 1995
Date Last Updated: May 2010 by Tim Daly
Description:
A package that implements the Brill-Noether algorithm. Part of the PAFF package.

GeneralPackageForAlgebraicFunctionField( K, symb, PolyRing, E, ProjPt, PCS, Plc, DIVISOR, InfClsPoint, DesTree, BLMET ) :Exports == Implementation where

K : Field
symb : List(Symbol)
OV == OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)
InfClsPoint : InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,BLMET)
DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory

FRACPOLY ==> Fraction PolyRing
OF ==> OutputForm
INT ==> Integer
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
UP ==> UnivariatePolynomial
UPZ ==> UP(t, Integer)
UTSZ ==> UnivariateTaylorSeriesCZero(Integer,t)
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)

ParamPackFC ==> LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_, E,ProjPt,PCS,Plc)
**PACKAGE GPAFF GENERALPACKAGEFORALGEBRAICFUNCTIONFIELD**

ParamPack   ==> ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
RatSingPack ==> ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
IntDivPack   ==> IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
IntFrmPack   ==> InterpolateFormsPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR)
DesTrPack    ==> DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
PackPoly     ==> PackageForPoly(K,PolyRing,E,#symb)

Exports ==> with

  reset: () -> Void
  setCurve: PolyRing -> PolyRing
  homogenize: (PolyRing,Integer) -> PolyRing
  printInfo: List Boolean -> Void
    ++ printInfo(lbool) prints some information coming from various
    ++ package and domain used by this package.
  theCurve: () -> PolyRing
    ++ theCurve returns the specified polynomial for the package.
  genus: () -> NNI
    ++ genus returns the genus of the curve defined by the polynomial
    ++ given to the package.
  genusNeg: () -> INT
  desingTree: () -> List DesTree
    ++ desingTree returns the desingularisation trees at all singular
    ++ points of the curve defined by the polynomial given to the package.
  desingTreeWoFullParam : () -> List Tree
    ++ desingTreeWoFullParam returns the desingularisation trees at all
    ++ singular points of the curve defined by the polynomial given to
    ++ the package. The local parameterizations are not computed.
  setSingularPoints: List ProjPt -> List ProjPt
    ++ setSingularPoints(lpt) sets the singular points to be used.
    ++ Beware: no attempt is made to check if the points are singular
    ++ or not, nor if all of the singular points are present. Hence,
    ++ results of some computation may be false. It is intended to be used
    ++ when one wants to compute the singular points are computed by other
    ++ means than to use the function singularPoints.
  singularPoints: () -> List(ProjPt)
    ++ rationalPoints() returns the singular points of the
+ curve defined by the polynomial given to the package.
+ If the singular points lie in an extension of the specified ground
+ field an error message is issued specifying the extension degree
+ needed to find all singular points.

parametrize: (PolyRing,Plc) -> PCS
+ parametrize(f,pl) returns a local parametrization of f at the place
+ pl.

lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
+ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> 
+ Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
+ interpolateForms(d,n) returns a basis of the interpolate forms of
+ degree n of the divisor d.

interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

eval: (PolyRing,Plc) -> K
+ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
+ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FRACPOLY,Plc) -> K
+ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
+ evalIfCan(f,pl) evaluate f at the place pl
+ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
+ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
+ (returns "failed" if it is a pole).

evalIfCan: (FRACPOLY,Plc) -> Union(K,"failed")
+ evalIfCan(u,pl) evaluate the function u at the place pl
+ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
+ intersectionDivisor(pol) compute the intersection divisor
+ (the Cartier divisor) of the form pol with the curve. If some
+ intersection points lie in an extension of the ground field,
+ an error message is issued specifying the extension degree
+ needed to find all the intersection points.
+ (If pol is not homogeneous an error message is issued).
adjunctionDivisor: () -> DIVISOR
++ adjunctionDivisor computes the adjunction divisor of the plane
++ curve given by the polynomial crv.

placesAbove: ProjPt -> List Plc

pointDominateBy : Plc -> ProjPt
++ pointDominateBy(pl) returns the projective point dominated
++ by the place pl.

if K has Finite then --should we say LocallyAlgebraicallyClosedField??

rationalPlaces: () -> List Plc
++ rationalPlaces returns all the rational places of the
++ curve defined by the polynomial given to the package.

rationalPoints: () -> List(ProjPt)

LPolynomial: () -> SparseUnivariatePolynomial Integer
++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

classNumber: () -> Integer
++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
++ placesOfDegree(d) returns all places of degree d of the
++ curve.

numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
++ places in the constant field extension of degree n

numberPlacesDegExtDeg: (PI, PI) -> Integer
++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of
++ degree n

ZetaFunction: () -> UTSZ
++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial

Implementation ==> add
import PPFC1
import PPFC2
import DesTrPack
import IntFrmPack
import IntDivPack
import RatSingPack
import ParamPack
import ParamPackFC
import PackPoly
crvLocal:PolyRing:=1$PolyRing
-- flags telling such and such is already computed.
genusCalc?:Boolean:= false()$Boolean
theGenus:INT:=0
desingTreeCalc?:Boolean:=false()$Boolean
theTree:List DesTree := empty()
desingTreeWoFullParamCalc?:Boolean:=false()$Boolean
adjDivCalc?:Boolean:=false()$Boolean
theAdjDiv:DIVISOR:=0
singularPointsCalc?:Boolean:=false()$Boolean
lesPtsSing:List(ProjPt):=empty()
rationalPointsCalc?:Boolean:=false()$Boolean
lesRatPts:List(ProjPt):=empty()
rationalPlacesCalc?:Boolean:=false()$Boolean
lesRatPlcs:List(Plc):=empty()
zf:UTSZ:=1$UTSZ
zfCalc : Boolean := false()$Boolean
DegOfPlacesFound: List Integer := empty()
-- see package IntersectionDivisorPackage
intersectionDivisor(pol)==
if ^(pol =$PolyRing homogenize(pol,1)) then _
  error _
  "From intersectionDivisor: the input is NOT a homogeneous polynomial"
intersectionDivisor(pol,theCurve(),desingTree(),singularPoints())

lBasis(divis)==
d := degree divis
if d < 0 then [[0$PolyRing], 1$PolyRing]
else A := adjunctionDivisor()

-- modifie le 08/05/97: avant c'était formToInterp := divOfZero(divis + A)
formToInterp := divOfZero(divis + A)

degDpA := degree formToInterp

degCrb := totalDegree(theCurve())$PackPoly
dd := divide(degDpA, degCrb pretend Integer)

dmin := NNI:
if ~zero?(dd.remainder) then (dd.quotient + 1) pretend NNI
else dd.quotient pretend NNI

print("Trying to interpolate with forms of degree:"::OF)
print(dmin::OF)
lg0 := interpolateForms(formToInterp, dmin)
while zero?(first lg0) repeat
  dmin := dmin + 1
  print("Trying to interpolate with forms of degree:"::OF)
  print(dmin::OF)
lg0 := interpolateForms(formToInterp, dmin)

print("Denominator found"::OF)
g0 := first lg0
dg0 := intersectionDivisor(g0)
print("Intersection Divisor of Denominator found"::OF)
lnumer := interpolateForms(dg0 - divis, dmin)
[lnumer, g0]

genus :=
if ~(genusCalc?) then
  degCrb := totalDegree(theCurve())$PackPoly
  theGenus := genusTreeNeg(degCrb, desingTreeWoFullParam())
  genusCalc := true()$Boolean

  if theGenus < 0 then
    print("Too many infinitly near points"::OF)
    print("The curve may not be absolutely irreducible"::OF)
    error "Have a nice day"

theGenus pretend NNI

genusNeg :=
if ~(genusCalc?) then
  degCrb := totalDegree(theCurve())$PackPoly
  theGenus := genusTreeNeg(degCrb, desingTreeWoFullParam())
  genusCalc := true()$Boolean

theGenus

homogenize(pol, n) := homogenize(pol, n)$PackPoly

fPl(pt: ProjPt, desTr: DesTree): Boolean ==
  nd := value desTr
  lpt := pointV nd
  pt = lpt
placesAbove(pt)==
  -- verifie si le point est simple, si c'est le cas,
  -- on retourne la place correspondante
  -- avec pointToPlace qui cree' la place si necessaire.
  ~member?(pt,singularPoints()) => _
  [pointToPlace(pt,theCurve())$ParamPackFC]
  -- les quatre lignes suivantes trouvent les feuilles qui
  -- sont au-dessus du point.
  theTree:= desingTree()
  cTree:= find(fPl(pt,#1),theTree)
  cTree case "failed" => error "Big error in placesAbove"
  -- G. Hache, gaetan.hache@inria.fr"
  lvs:=leaves cTree
  -- retourne les places correspondant aux feuilles en "consultant"
  -- les diviseurs exceptionnels.
  concat [supp excpDivV(l) for l in lvs]

pointDominateBy(pl)== pointDominateBy(pl)$ParamPackFC

reduceForm(p1:PolyRing,p2:PolyRing):PolyRing==
  normalForm(p1,[p2])$GroebnerPackage(K,E,OV,PolyRing)

evalIfCan(f:PolyRing,pl:Plc)==
  u:=reduceForm(f, theCurve() )
  zero?(u) => 0
  pf:= parametrize(f,pl)
  ord:INT:=order pf
  ord < 0 => "failed"
  ord > 0 => 0
  coefOfFirstNonZeroTerm pf

eval(f:PolyRing,pl:Plc)==
  eic:=evalIfCan(f,pl)
  eic case "failed" => _
  error "From eval (function at place): its a pole !!!"
  eic

setCurve(pol)==
  crvLocal:=pol
  "(crvLocal =$PolyRing homogenize(crvLocal,1)) =>
  print("the defining polynomial is not homogeneous")::OF)
  error "Have a nice day"
  reset()
  theCurve()

reset ==
  setFoundPlacesToEmpty()$Plc
  genusCalc?:Boolean:= false()$Boolean
theGenus: INT := 0

desingTreeCalc?: Boolean := false()$Boolean

desingTreeWoFullParamCalc?: Boolean := false()$Boolean

theTree: List DesTree := empty()

adjDivCalc?: Boolean := false()$Boolean

theAdjDiv: DIVISOR := 0

singularPointsCalc?: Boolean := false()$Boolean

lesPtsSing: List ProjPt := empty()

rationalPointsCalc?: Boolean := false()$Boolean

lesRatPts: List ProjPt := empty()

rationalPlacesCalc?: Boolean := false()$Boolean

lesRatPlcs: List Plc := empty()

DegOfPlacesFound: List Integer := empty()

zf: UTSZ := 1$UTSZ

zfCalc: Boolean := false$Boolean

foundPlacesOfDeg?(i: PositiveInteger): Boolean ==

ld: List Boolean := [zero?(a rem i) for a in DegOfPlacesFound]

tree?(true$Boolean, ld)

findOrderOfDivisor(divis, lb, hb) ==

"zero?(degree divis) => error("The divisor is NOT of degree zero !!!!")"

A := adjunctionDivisor()

formToInterp := divOfZero( hb*divis + A )

degDpA := degree formToInterp

degCrb := totalDegree( theCurve())$PackPoly

dd := divide(degDpA, degCrb pretend Integer)

dmin: NNI :=

if ~zero?(dd.remainder) then (dd.quotient+1) pretend NNI
  else dd.quotient pretend NNI

lg0: List PolyRing := interpolateForms(formToInterp, dmin)

while zero?(first lg0) repeat

dmin := dmin+1

lg0 := interpolateForms(formToInterp, dmin)

g0: PolyRing := first lg0

dg0 := intersectionDivisor(g0)

nhb := hb

while effective?(dg0 - nhb*divis - A) repeat

  nhb := nhb+1

ftry := 1b

lnumer: List PolyRing := interpolateForms(dg0-ftry*divis, dmin)

while zero?(first lnumer) and ftry < nhb repeat

  ftry := ftry + 1

  lnumer := interpolateForms(dg0-ftry*divis, dmin)

[ftry, first lnumer, g0, nhb]

theCurve ==

one?(crvLocal) => error "The defining polynomial has not been set yet!"

crvLocal
printInfo(lbool) ==
  printInfo(lbool.2)$ParamPackFC
printInfo(lbool.3)$PCS
void()

desingTree ==
  theTree := desingTreeWoFullParam()
  if ~(desingTreeCalc?) then
    for arb in theTree repeat
      fullParamInit(arb)
      desingTreeCalc? := true()$Boolean
    theTree
  desingTreeWoFullParam ==
  if ~(desingTreeWoFullParamCalc?) then
    theTree := [desingTreeAtPoint(pt, theCurve()) for pt in singularPoints()]
    desingTreeWoFullParamCalc? := true()$Boolean
    theTree
  adjunctionDivisor() ==
  if ~(adjDivCalc?) then
    theAdjDiv := _
    reduce("+", [adjunctionDivisor(tr) for tr in desingTree()], 0$DIVISOR)
    adjDivCalc? := true()$Boolean
    theAdjDiv

  singularPoints() ==
  if ~(singularPointsCalc?) then
    lesPtsSing := singularPoints(theCurve())
    singularPointsCalc? := true()$Boolean
    lesPtsSing

  setSingularPoints(lspt) ==
  singularPointsCalc? := true()$Boolean
  lesPtsSing := lspt

  interpolateForms(d, n) ==
  -- compute the local parametrization of f at the place pl
  -- (from package ParametrizationPackage)
  parametrize(f, pl) == parametrize(f, pl)$ParamPack

  interpolateForms(d, n) ==
lm: List PolyRing := listAllMono(n)$PackPoly
interpolateForms(d,n, theCurve(), lm)

interpolateFormsForFact(d, lm)==
interpolateFormsForFact(d, lm)$IntFrmPack

evalIfCan(f: PolyRing, g: PolyRing, pl: Plc)==
  fu := reduceForm(f, theCurve())
  gu := reduceForm(g, theCurve())
  zero?(fu) and zero?(gu) => 0
  zero?(fu) and ^zero?(gu) => "failed"
  pf := parametrize(fu, pl)
  pg := parametrize(gu, pl)
  ordf: INT := order pf
  ordg: INT := order pg
  cf := coefOfFirstNonZeroTerm pf
  cg := coefOfFirstNonZeroTerm pg
  (ordf - ordg) < 0 => "failed"
  (ordf - ordg) > 0 => 0
  cf * inv cg

eval(f: PolyRing, g: PolyRing, pl: Plc)==
  eic := evalIfCan(f, g, pl)
  eic case "failed" => error "From eval (function at place): its a pole"
  eic

evalIfCan(u: FRACPOLY, pl: Plc)==
  f: PolyRing := numer u
  g: PolyRing := denom u
  evalIfCan(f, g, pl)

eval(u: FRACPOLY, pl: Plc)==
  f: PolyRing := numer u
  g: PolyRing := denom u
  eval(f, g, pl)

thedeg: PI := 1

crap(p: Plc): Boolean ==
  degree(p)$Plc = thedeg

if K has Finite then
  rationalPlaces ==
    K has PseudoAlgebraicClosureOfFiniteFieldCategory => _
    placesOfDegree(1)$PI
    -- non good pour LACF !!!!
  rationalPlacesCalc? => lesRatPlcs
  ltr: List(DesTree):= desingTree()
  ratP: List(ProjPt):= rationalPoints()
  singP: List(ProjPt):= singularPoints()
simRatP:List(ProjectPt):=setDifference(ratP,singP)
for pt in simRatP repeat
    pointToPlace(pt,theCurve())$ParamPackFC
rationalPlacesCalc? := true()$Boolean
lesRatPlcs:=foundPlaces()$Plc
lesRatPlcs

rationalPoints==
    if ~(rationalPointsCalc?) then
        if K has Finite then
            lesRatPts:= rationalPoints(theCurve(),1)$RatSingPack
            rationalPointsCalc?:=true()$Boolean
        else
            error "Can't find rationalPoints when the field is not finite"
        lesRatPts
    ZetaFunction() ==
    if not zfCalc then
        zf:= ZetaFunction(1)
        zfCalc:= true$Boolean
    zf

ZetaFunction(d) ==
    lp:= LPolynomial(d)
    if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
        setTower!(1$K)
    q:INT := size()$K ** d
    lpt:UPZ := unmakeSUP(lp)$UPZ
    lps:UTSZ := coerce(lpt)$UTSZ
    x:= monomial(1,1)$UTSZ
    mul: UTSZ := (1-x)*(1 - q * x)
    invmul:Union(UTSZ,"failed") := recip(mul)$UTSZ
    ivm: UTSZ
    if not (invmul case "failed") then
        ivm := invmul pretend UTSZ
    else
        ivm := 1
        lps * ivm
    calculatedSer: List UTSZ:= [1]
    --in index i is the "almost ZetaFunction" summed to i-1.
    --Except calculatedSer.1 which is 1
    numberOfPlacesOfDegreeUsingZeta(degree:PI): Integer ==
    --is at most called once for each degree. Will calculate the
    --entries in calculatedSer.
    ser:UTSZ := 1
    x:= monomial(1,1)$UTSZ
    pol:UTSZ
    polser:Union(UTSZ,"failed")
serdel := UTSZ
i := maxIndex(calculatedSer) pretend PI
while i < degree repeat
    serdel := 1
    if (n := numberOfPlacesOfDegree(i)) > 0 then
        pol := (1 - x**i)**(n pretend PI)
        polser := recip(pol)$UTSZ -- coerce(pol)$UTSZ)$UTSZ
        if not (polser case "failed") then
            serdel := (polser pretend UTSZ)
        else
            error "In numberOfPlacesOfDegreeUsingZeta. This shouldn't happen"
    ser := serdel * calculatedSer.i
    calculatedSer := concat(calculatedSer, ser)
    i := i + 1
if degree = 1 then
    coefficient(ZetaFunction(), degree)
else
    coefficient(ZetaFunction(), degree) - _
    coefficient(calculatedSer.degree, degree)

numberOfPlacesOfDegree(i:PI): Integer ==
    if zfCalc then
        if (m := maxIndex(calculatedNP)) < i then
            calculatedNP := _
            concat(calculatedNP, _
            [numberOfPlacesOfDegreeUsingZeta(j pretend PI) _
             for j in ((m+1) pretend PI)..i])
        calculatedNP.i
    else
        # placesOfDegree(i) -- maybe we should make an improvement in this
        placesOfDegree(i) ==
        if (not foundPlacesOfDeg?(i)) then
            if characteristic()$K**i > (2**16 - 1) then
                print("If you are using a prime field and"::OF)
                print("GB this will not work."::OF)
            desingTree()
            placesOfDegree(i, theCurve(), singularPoints())
        DegOfPlacesFound := concat(DegOfPlacesFound, i)
        thedeg := i
        select(crap(#1), foundPlaces()$Plc)
    numberRatPlacesExtDeg(extDegree:PI): Integer ==
    numberPlacesDegExtDeg(1, extDegree)

numberPlacesDegExtDeg(degree:PI, extDegree:PI): Integer ==
    res: Integer := 0
m:PI := degree * extDegree
d: PI
while m > 0 repeat
  d:= gcd(m, extDegree)
  if (m quo d) = degree then
    res:= res + (numberOfPlacesOfDegree(m) * d)
    m:= (m - 1) pretend PI
res

calculateS(extDeg:PI): List Integer ==
g := genus()
sizeK:NNI := size($K ** extDeg
i:PositiveInteger := g pretend PI
S: List Integer := [0 for j in 1..g]
good:Boolean := true()$Boolean
while good repeat
  S.i := numberRatPlacesExtDeg(i*extDeg) - ((sizeK **$NNI i) + 1)
  j:Integer := i - 1
  if (not (j = 0)) then
    i:= (j pretend PI)
  else good:= false()$Boolean
S

LPolynomial(): SparseUnivariatePolynomial Integer ==
LPolynomial(1)

LPolynomial(extDeg:PI): SparseUnivariatePolynomial Integer ==
--when translating to AxiomXL rewrite this function!
g := genus()
zero?(g) => 1
coef: List Integer := [1]
if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
  setTower!(1$K)
  sizeK:Integer := size($K ** extDeg --need to do a setExtension before
  coef:= concat(coef,[0 for j in 1..(2*g)])
S: List Integer := calculateS(extDeg)
i:PI := 1
tmp:Integer
  while i < g + 1 repeat
    j:PI := 1
tmp:= 0
  while j < i + 1 repeat
    tmp:= tmp + S.j * coef((i + 1 - j) pretend PI)
    j:= j + 1
    coef.(i+1) := tmp quo i
    i:= i + 1
    i:= 1
  while i < g + 1 repeat
    ss: Integer := sizeK **$Integer ((g + 1 - i) pretend PI)
    val:Integer := ss * coef.i
coef.((2*g+2 - i) pretend PI) := val
i:= i + 1
x:= monomial(1,1)$SUP(INT)
   result: SparseUnivariatePolynomial(Integer):= _
      1$SparseUnivariatePolynomial(Integer)
coef:= rest(coef)
i:= 1
while i < 2 * g + 1 repeat
   pol: SUP(INT) := (first(coef) :: Integer) * (x ** i)
      result:= result + pol --(first(coef) :: Integer) * (x ** i)
      coef:= rest(coef)
i:= i + 1
      result
classNumber():Integer ==
   LPolynomial()$(1)

—— GPAFF.dotabb ——

"GPAFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GPAFF"]
"DTP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DTP"]
"INTDIVP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTDIVP"]
"GPAFF" -> "INTDIVP"
"GPAFF" -> "DTP"

———

package GENPGCD GeneralPolynomialGcdPackage

—— GeneralPolynomialGcdPackage.input ——

)set break resume
)sys rm -f GeneralPolynomialGcdPackage.output
)spool GeneralPolynomialGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralPolynomialGcdPackage
--E 1
GeneralPolynomialGcdPackage help

This package provides operations for GCD computations on polynomials

See Also:
o )show GeneralPolynomialGcdPackage

Exports:
gcdPolynomial randomR

package GENPGCD GeneralPolynomialGcdPackage
E : OrderedAbelianMonoidSup
SUPP ==> SparseUnivariatePolynomial P
--JHD ContPrim ==> Record(cont:P,prim:P)

C == with
  gcdPolynomial : (SUPP,SUPP) -> SUPP
  ++ gcdPolynomial(p,q) returns the GCD of p and q
  randomR : () -> R
  ++ randomR() should be local but conditional
--JHD gcd : (P,P) -> P
--JHD gcd : List P -> P
--JHD gcdprim : (P,P) -> P
--JHD gcdprim : List P -> P
--JHD gcdcofact : List P -> List P
--JHD gcdcofactprim : List P -> List P
--JHD primitate : (P,OV) -> P
--JHD primitate : SUPP -> SUPP
--JHD content : P -> P
--JHD content : List P -> List P
--JHD contprim : List P -> List ContPrim
--JHD monomContent : (P,OV) -> P
--JHD monomContent : SUPP -> SUPP

T == add

SUPR ==> SparseUnivariatePolynomial R
--JHD SUPLGcd ==> Record(locgcd:SUPP,goodint:List R)
--JHD Lgcd ==> Record(locgcd:P,goodint:List R)
--JHD UTerm ==> Record(lpol:List SUPR,lint:List R,mpol:P)
--JHD--JHD pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

--JHD import MultivariateLifting(E,OV,R,P,pmod)
import UnivariatePolynomialCategoryFunctions2(R,SUPR,P,SUPP)
import UnivariatePolynomialCategoryFunctions2(P,SUPP,R,SUPR)

-------- Local Functions --------
--JHD abs : P -> P
better : (P,P) -> Boolean
--JHD failtest : (P,P,P) -> Boolean
--JHD gcdMonom : (P,P,OV) -> P
--JHD gcdTermList : (P,P) -> P
--JHD gcdPrim : (P,P,OV) -> P
--JHD gcdSameMainvar : (P,P,OV) -> P
--JHD internal : (P,P,OV) -> P
--JHD  good          : (P,List OV) -> Record(upol:SUPR,inval:List R)
--JHD  gcdPrs        : (P,P,NNI,OV) -> Union(P,"failed")
--JHD
--JHD  chooseVal     : (P,P,List OV) -> UTerm
--JHD  localgcd      : (P,P,List OV) -> LGcd
--JHD  notCoprime    : (P,P, List NNI,List OV) -> P
--JHD  imposelc      : (List SUPR,List OV,List R,List P) -> List SUPR

--JHD  lift?              : (P,P,UTerm,List NNI,List OV) -> Union("failed",P)
--  lift             : (P,SUPR,SUPR,P,List OV,List NNI,List R) -> P
  -- lifts first and third arguments as factors of the second
  -- fourth is number of variables.
--JHD  monomContent    : (P,OV) -> P
--  monomContentSup   : SUPP -> SUPP
--
--JHD  gcdcofact      : List P -> List P

gcdTrivial     : (SUPP,SUPP) -> SUPP
gcdSameVariables: (SUPP,SUPP,List OV) -> SUPP
recursivelyGCDCoefficients: (SUPP,List OV,SUPP,List OV) -> SUPP
flatten        : (SUPP,List OV) -> SUPP
  -- evaluates out all variables in the second
  -- argument, leaving a polynomial of the same
  -- degree
--  eval             : (SUPP,List OV,List R) -> SUPP
  variables        : SUPP -> List OV
---- JHD’s exported functions ---
gcdPolynomial(p1:SUPP,p2:SUPP) ==
  zero? p1 => p2
  zero? p2 => p1
  0=degree p1 => gcdTrivial(p1,p2)
  0=degree p2 => gcdTrivial(p2,p1)
  if degree p1 < degree p2 then (p1,p2):=(p2,p1)
  p1 exquo p2 case SUPP => (unitNormal p2).canonical
  c1:= monomContentSup(p1)
  c2:= monomContentSup(p2)
  p1:=(p1 exquo c1)::SUPP
  p2:=(p2 exquo c2)::SUPP
  (p1 exquo p2) case SUPP => (unitNormal p2).canonical * gcd(c1,c2)
v1:=variables p1
v2:=variables p2
v1:=setDifference(v1,v2)
v2:=setDifference(v2,v1)
if #v1 = 0 and #v2 = 0 then gcdSameVariables(p1,p2,v1,v2)
  -- all variables are in common
v:=setDifference(v1,v2)
p1:=flatten(p1,v1)
p2:=flatten(p2,v2)
g:=gcdSameVariables(p1,p2,v)
one? g => gcd(c1,c2)::SUPP
(g = 1) => gcd(c1,c2)::SUPP

(#v1 = 0 or not (p1 exquo g) case "failed") and
  -- if #vi = 0 then pp1 = p1, so we know g divides
  (#v2 = 0 or not (p2 exquo g) case "failed")
  => g*gcd(c1,c2) -- divides them both, so is the gcd

v:=variables g -- there can be at most these variables in answer
v1:=setDifference(vp1,v)
v2:=setDifference(vp2,v)
if (#v1 = 0) then g:=gcdSameVariables(g,flatten(p2,v2),v)
else if (#v2=0) then g:=gcdSameVariables(g,flatten(p1,v1),v)
else g:=gcdSameVariables(g,flatten(p1,v1)-flatten(p2,v2),v)

one? g => gcd(c1,c2)::SUPP
(g = 1) => gcd(c1,c2)::SUPP

v:=variables g -- there can be at most these variables in answer
v1:=setDifference(vp1,v)
if #v1 ^= 0 then
  g:=recursivelyGCDCoefficients(g,v,p1,v1)

one? g => return gcd(c1,c2)::SUPP
(g = 1) => return gcd(c1,c2)::SUPP

v:=variables g -- there can be at most these variables in answer
v2:=setDifference(vp2,v)
recursivelyGCDCoefficients(g,v,p2,v2)*gcd(c1,c2)

if R has StepThrough then
  randomCount:R := init()
  randomR() ==
    (v:=nextItem(randomCount)) case R =>
      randomCount:=v
      v
      SAY("Taking next stepthrough range in GeneralPolynomialGcdPackage")$Lisp
      randomCount:=init()
      randomCount
  else
    randomR() == (random$Integer() rem 100)::R
    gcdSameVariables(p1:SUPP,p2:SUPP,lv:List OV) ==
      -- two non-trivial primitive (or, at least, we don't care
      -- about content)
      -- polynomials with precisely the same degree
      #lv = 0 => map((x:R):P+->x::P,gcdPolynomial(map(ground,p1),
          map(ground,p2)))
      degree p2 = 1 =>
        p1 exquo p2 case SUPP => p2
        1
      gcdLC:=gcd(leadingCoefficient p1,leadingCoefficient p2)
      lr:=[randomR() for vv in lv]
count: NonNegativeInteger := 0
while count < 10 repeat
  while zero? eval(gcdLC, lv, lr) and count < 10 repeat
    lr := [randomR() for vv in lv]
    count := count + 1
  count = 10 => error "too many evaluations in GCD code"
  up1 : SUPR := map(y +-> ground eval(y, lv, lr), p1)
  up2 : SUPR := map(x +-> ground eval(x, lv, lr), p2)
  u := gcdPolynomial(up1, up2)
  degree u = 0 => return 1
  -- let's pick a second one, just to check
  lrr := [randomR() for vv in lv]
  while zero? eval(gcdLC, lv, lrr) and count < 10 repeat
    lrr := [randomR() for vv in lv]
    count := count + 1
  count = 10 => error "too many evaluations in GCD code"
  vp1 : SUPR := map(x +-> ground eval(x, lv, lrr), p1)
  vp2 : SUPR := map(y +-> ground eval(y, lv, lrr), p2)
  v := gcdPolynomial(vp1, vp2)
  degree v = 0 => return 1
  if degree v < degree u then
    u := v
    up1 := vp1
    up2 := vp2
    lr := lrr
    up1 := (up1 exquo u)::SUPR
    degree gcd(u, up1) = 0 =>
      ans := lift(u, p1, up1, lv, lr)
      ans case SUPP => return ans
      "next"
    up2 := (up2 exquo u)::SUPR
    degree gcd(u, up2) = 0 =>
      ans := lift(u, p2, up2, lv, lr)
      ans case SUPP => return ans
      "next"
  -- so neither cofactor is relatively prime
  count := 0
  while count < 10 repeat
    r := randomR()
    uu := up1 + r * up2
    degree gcd(u, uu) = 0 =>
      ans := lift(u, p1 + r::P, uu, lv, lr)
      ans case SUPP => return ans
      "next"
  error "too many evaluations in GCD code"
  count >= 10 => error "too many evaluations in GCD code"
  lift(gR: SUPR, p: SUPP, cfR: SUPR, lv: List OV, lr: List R) ==
  -- lift the coprime factorisation gR*cfR = (univariate of p)
  -- where the variables lv have been evaluated at lr
  lcp := leadingCoefficient p
g := monomial(lcp, degree gR) + map(x -> x::P, reductum gR)
cf := monomial(lcp, degree cfR) + map(y -> y::P, reductum cfR)
p := lcp*p -- impose leading coefficient of p on each factor
while lv ^= [] repeat
  v := first lv
  r := first lr
  lv := rest lv
  lr := rest lr
  thisp := map(x1 -> eval(x1, lv, lr), p)
  d := "max"/[degree(c, v) for c in coefficients p]
  prime := v::P - r::P
  pn := prime
  origFactors := [g, cf]::List SUPP
  for n in 1..d repeat
    Ecart := (thisp - g*cf) exquo pn
    Ecart case "failed" =>
      error "failed lifting in hensel in Complex Polynomial GCD"
    zero? Ecart => leave
    step := solveLinearPolynomialEquation(origFactors,
                                           map(x2 -> eval(x2, v, r), Ecart::SUPP))
    step case "failed" => return "failed"
    g := g + pn*first step
    cf := cf + pn*second step
    pn := pn*prime
    thisp ^= g*cf => return "failed"
g
recursivelyGCDCoefficients(g:SUPP, v: List OV, p:SUPP, pv: List OV) ==
  mv := first pv -- take each coefficient w.r.t. mv
  pv := rest pv -- and recurse on pv as necessary
  d := "max"/[degree(u, mv) for u in coefficients p]
  for i in 0..d repeat
    p1 := map(x -> coefficient(x, mv, i), p)
    oldg := g
    if pv = [] then g := gcdSameVariables(g, p1, v)
    else g := recursivelyGCDCoefficients(p, v, p1, pv)
      -- one? g => return 1
      (g = 1) => return 1
      g := oldg =>
        oldv := v
        v := variables g
        pv := setUnion(pv, setDifference(v, oldv))
  g
flatten(p1:SUPP, lv: List OV) ==
  #lv = 0 => p1
  lr := [randomR() for vv in lv]
  dg := degree p1
  while dg ^= degree (ans := map(x -> eval(x, lv, lr), p1)) repeat
    lr := [randomR() for vv in lv]
  ans
  -- eval(p1:SUPP, lv: List OV, lr: List R) == map(eval(#1, lv, lr), p1)
variables(p1:SUPP) ==
  removeDuplicates ("concat"/variables u for u in coefficients p1)

gcdTrivial(p1:SUPP,p2:SUPP) ==
  -- p1 is non-zero, but has degree zero
  -- p2 is non-zero
  cp1:=leadingCoefficient p1
  -- one? cp1 => 1
  (cp1 = 1) => 1
  degree p2 = 0 => gcd(cp1,leadingCoefficient p2)::SUPP
  un?:=unit? cp1
  while not zero? p2 and not un? repeat
    cp1:=gcd(leadingCoefficient p2,cp1)
    un?:=unit? cp1
    p2:=reductum p2
  un? => 1
  cp1::SUPP

---- Local functions ----
--JHD -- test if something wrong happened in the gcd
--JHD (p1 exquo f) case "failed" or (p2 exquo f) case "failed"
--JHD
--JHD -- Choose the integers
--JHD x:OV:=lvar.first
--JHD lvr:=lvar.rest
--JHD d1:=degree(p1,x)
--JHD d2:=degree(p2,x)
--JHD dd:NNI:=0$NNI
--JHD nvr:NNI:=#lvr
--JHD lval:List R :=[]
--JHD range:I:=8
--JHD for i in 1.. repeat
  --JHD range:=2*range
  --JHD lval:=[(random()$I rem (2*range) - range)::R for i in 1..nvr]
--JHD uf1:SUPR:=univariate eval(p1,lvr,lval)
--JHD degree uf1 ^= d1 => "new point"
--JHD uf2:SUPR:=univariate eval(p2,lvr,lval)
--JHD degree uf2 ^= d2 => "new point"
--JHD u:=gcd(uf1,uf2)
--JHD du:=degree u
--JHD --the univariate gcd is 1
--JHD if du=0 then return [[1$SUPR],lval,0$P]$UTerm
--JHD
--JHD ugcd:List SUPR:=[u,(uf1 exquo u)::SUPR,(uf2 exquo u)::SUPR]
--JHD uterm:=[ugcd,lval,0$P]$UTerm
--JHD dd=0 => dd:=du
--JHD
--JHD --the degree is not changed
--JHD du=dd =>
--JHD --test if one of the polynomials is the gcd
--JHD dd=d1 =>
--JHD if "((f:=p2 exquo p1) case "failed") then
--JHD return [[u],lval,p1]$UTerm
--JHD if dd=d2 then dd:=(dd-1)::NNI
--JHD
--JHD dd=d2 =>
--JHD if "((f:=p1 exquo p2) case "failed") then
--JHD return [[u],lval,p2]$UTerm
--JHD dd:=(dd-1)::NNI
--JHD return uterm
--JHD
--JHD --the new gcd has degree less
--JHD du<dd => dd:=du
--JHD
--JHD good(f:P,lvr:List OV):Record(upol:SUPR,inval:List R) ==
--JHD nvr:NNI:=#lvr
--JHD range:I:=1
--JHD ltry:List List R:=[]
--JHD while true repeat
--JHD range:=2*range
--JHD lval:=[(random()$I rem (2*range) -range)::R for i in 1..nvr]
--JHD member?(lval,ltry) => "new point"
--JHD ltry:=cons(lval,ltry)
--JHD uf:=univariate eval(f,lvr,lval)
--JHD if degree gcd(uf,differentiate uf)=0 then return [uf,lval]
--JHD
--JHD -- impose the right lc
--JHD imposelc(lipol:List SUPR,
--JHD result:List SUPR :=[]
--JHD lvar:=lvar.rest
--JHD for pol in lipol for leadpol in leadc repeat
--JHD p1:= univariate eval(leadpol,lvar,lval) * pol
--JHD result:= cons((p1 exquo leadingCoefficient pol):SUPR,result)
--JHD reverse result
--JHD
--JHD --Compute the gcd between not coprime polynomials
--JHD notCoprime(g:P,p2:P,ldeg:List NNI,lvar:List OV) : P ==
--JHD x:OV:=lvar.first
--JHD lvar1:List OV:=lvar.rest
--JHD lg1:=gcdcofact([g,differentiate(g,x)])
--JHD g1:=lg1.1
--JHD lg:LGcd:=localgcd(g1,p2,lvar)
--JHD (l,1lval):=(lg.locgcd,lg.goodint)
--JHD p2:=(p2 exquo l)::P
--JHD (gd1,gd2):=(1,1)
--JHD ul:=univariate(eval(l,lvar1,lval))
--JHD d1:=degree ul
--JHD if degree gcd(ul,differentiate ul) ^=0 then
--JHD newchoice:=good(lvar,rest)
--JHD ul:=newchoice.upol
--JHD lval:=newchoice.inval
--JHD ug1:=univariate(eval(g1,lvar,lval))
--JHD ulist:=[ug1,univariate eval(p2,lvar,lval)]
--JHD lcpol:=[leadingCoefficient univariate(g1,x),
       leadingCoefficient univariate(p2,x)]
--JHD while true repeat
--JHD d:SUPR:=gcd(cons(ul,ulist))
--JHD if degree d =0 then return gd1
--JHD lquo:=(ul exquo d)::SUPR
--JHD if degree lquo ^=0 then
--JHD lgcd:=gcd(cons(leadingCoefficient univariate(l,x),lcpol))
--JHD gd2:=lift(1,d,lquo,lgcd,lvar,1deg,lval)
--JHD l:=gd2
--JHD ul:=univariate(eval(l,lvar,lval))
--JHD dl:=degree ul
--JHD gd1:=gd1*gd2
--JHD ulist:=[(uf exquo d)::SUPR for uf in ulist]
--JHD
--JHD -- we suppose that the poly have the same mainvar, deg p1<deg p2 and the
--JHD -- polys primitive
--JHD internal(p1:P,p2:P,x:OV) : P ==
--JHD lvar:List OV:=sort(#1>#2,setUnion(variables p1,variables p2))
--JHD d1:=degree(p1,x)
--JHD d2:=degree(p2,x)
--JHD result: P:=localgcd(p1,p2,lvar).locgcd
--JHD -- special cases
--JHD result=1 => 1$P
--JHD (dr:=degree(result,x))=d1 or dr=d2 => result
--JHD while failtest(result,p1,p2) repeat
--JHD SAY$Lisp "retrying gcd"
--JHD result:=localgcd(p1,p2,lvar).locgcd
--JHD result
--JHD
--JHD --local function for the gcd : it returns the evaluation point too
--JHD localgcd(p1:P,p2:P,lvar:List(OV)) : LGcd ==
--JHD x:OV:=lvar.first
--JHD uterm:=chooseVal(p1,p2,lvar)
--JHD listpol:= uterm.lpol
--JHD ud:=listpol.first
--JHD dd:= degree ud
--JHD
--JHD --the univariate gcd is 1
--JHD dd=0 => [1$P,uterm.lint]$LGcd
--JHD
--JHD --one of the polynomials is the gcd
--JHD dd=degree(p1,x) or dd=degree(p2,x) =>
--JHD [uterm.mpol,uterm.lint]$LGcd
--JHD  ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))
--JHD
--JHD  -- if there is a polynomial g s.t. g/gcd and gcd are coprime ...
--JHD  -- I can lift
--JHD  (h:=lift?(p1,p2,uterm,ldeg,lvar)) case "failed" =>
--JHD  [notCoprime(p1,p2,ldeg,lvar),uterm.lint]$LGcd
--JHD  [h::P,uterm.lint]$LGcd
--JHD
--JHD
--JHD  -- content, internal functions return the poly if it is a monomial
--JHD  monomContent(p:P,var:OV):P ==
--JHD  ground? p => 1$P
--JHD  md:= minimumDegree(p,var)
--JHD  ((var::P)**md)*(gcd sort(better,coefficients univariate(p,var)))

--JHD  -- change the polynomials to have positive lc
--JHD  abs(p:P): P == unitNormal(p).canonical

-- Ordering for gcd purposes
better(p1:P,p2:P):Boolean ==
  ground? p1 => true
  ground? p2 => false
  degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

-- PNS algorithm
--  ul:= univariate(p1,var)
--  u2:= univariate(p2,var)
--  finished:Boolean:= false
--  until finished repeat
--    dd:NNI:=(degree ul - degree u2):NNI
--    1c1:SUFP:=leadingCoefficient u2 * reductum u1
--    1c2:SUFP:=leadingCoefficient u1 * reductum u2
--    u3:SUFP:= primitate((1c1-1c2)*monomial(1$P,dd))%
--    (d3:=degree(u3)) <= d => finished:= true
--    ul:= u2
--    u2:= u3
--    if d3 > degree(u1) then (u1,u2):= (u2,u1)
--    g:=(u2 x quo u3)
--    g case SUPP => abs multivariate(u3,var)
--    "failed"

-- Gcd between polynomial p1 and p2 with
-- mainVariable p1 < x=mainVariable p2
--JHD  gcdTermList(p1:P,p2:P) : P ==
CHAPTER 8. CHAPTER G

---JHD termList:=sort(better,
---JHD cons(p1,coefficients univariate(p2,(mainVariable p2)::OV)))
---JHD q:P:=termList.first
---JHD for term in termList.rest until q = 1$P repeat q:= gcd(q,term)
---JHD q
---JHD

---JHD -- Gcd between polynomials with the same mainVariable
---JHD if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
---JHD (p1 exquo p2) case P => abs p2
---JHD c1:= monomContent(p1,mvar)$%
---JHD c1 = p1 => gcdMonom(p1,p2,mvar)
---JHD c2:= monomContent(p2,mvar)$%
---JHD c2 = p2 => gcdMonom(p2,p1,mvar)
---JHD p1:= (p1 exquo c1)::P
---JHD p2:= (p2 exquo c2)::P
---JHD if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
---JHD (p1 exquo p2) case P => abs(p2) * gcd(c1,c2)
---JHD abs(gcdPrim(p1,p2,mvar)) * gcd(c1,c2)
---JHD

---JHD -- make the polynomial primitive with respect to var
---JHD primitate(p:P,var:OV):P == (p exquo monomContent(p,var))::P
---JHD
---JHD primitate(u:SUPP):SUPP == (u exquo monomContentSup u)::SUPP
---JHD

---JHD -- gcd between primitive polynomials with the same mainVariable
---JHD vars:= removeDuplicates append(variables p1,variables p2)
---JHD #vars=1 => multivariate(gcd(univariate p1,univariate p2),mvar)
---JHD vars:=delete(vars,position(mvar,vars))
---JHD --d:= degModGcd(p1,p2,mvar,vars)
---JHD --d case "failed" => internal(p2,p1,mvar)
---JHD --deg:= d:NNI
---JHD --deg = 0$NNI => 1$P
---JHD --deg = degree(p1,mvar) =>
---JHD -- (p2 exquo p1) case P => abs(p1) -- already know that
---JHD --  "(p1 exquo p2)
---JHD -- internal(p2,p1,mvar)
---JHD --cheapPrs?(p1,p2,deg,mvar) =>
---JHD -- g:= gcdPrs(p1,p2,deg,mvar)
---JHD -- g case P => g::P
---JHD -- internal(p2,p1,mvar)
---JHD internal(p2,p1,mvar)
---JHD

---JHD -- gcd between a monomial and a polynomial
---JHD ((var::P) ** min(minimumDegree(m,ov),minimumDegree(p,ov))) *
---JHD gcdTermList(leadingCoefficient(univariate(m,ov)),p)
---JHD

---JHD -- If there is a pol s.t. pol/gcd and gcd are coprime I can lift
--JHD lift?(p1:P,p2:P,uterm:UTerm,ldeg:List NNI,
--JHD   lvar:List OV) : Union("failed",P) ==
--JHD   x:OV:=lvar.first
--JHD   leadpol:Boolean:=false
--JHD   (listpol,lval):=(uterm.lpol,uterm.lint)
--JHD   d:=listpol.first
--JHD   listpol:=listpol.rest
--JHD   nolift:Boolean:=true
--JHD   for uf in listpol repeat
--JHD     --note uf and d not necessarily primitive
--JHD     degree gcd(uf,d) =0 => nolift:=false
--JHD     nolift => "failed"
--JHD   f:P:=[p1,p2]$List(P).position(uf,listpol))
--JHD   lgcd:=gcd(leadingCoefficient univariate(p1,x),
--JHD     leadingCoefficient univariate(p2,x))
--JHD   lift(f,d,uf,lgcd,lvar,ldeg,lval)
--JHD
--JHD -- interface with the general "lifting" function
--JHD lift(f:P,d:SUPR,uf:SUPR,lgcd:P,lvar:List OV,
--JHD   ldeg:List NNI,lval:List R):P ==
--JHD   x:OV:=lvar.first
--JHD   leadpol:Boolean:=false
--JHD   lcf:P
--JHD   lcf:=leadingCoefficient univariate(f,x)
--JHD   df:=degree(f,x)
--JHD   leadlist:List(P):=[
--JHD     if lgcd=1$P then
--JHD     leadpol:=true
--JHD     f:=lgcd*f
--JHD     ldeg:=[n0+n1 for n0 in ldeg for n1 in degree(lgcd,lvar)]
--JHD     lcd:R:=leadingCoefficient d
--JHD     if ground? lgcd then d:=(retract(lgcd)*d exquo lcd)::SUPR
--JHD     else d:=(eval(lgcd,lvar.rest,lval)) * d exquo lcd)::SUPR
--JHD     uf:=1*uf
--JHD     leadlist:=[lgcd,lcf]
--JHD     lg:=imposeic([d,uf],lvar,lval,leadlist)
--JHD     plist:=lifting(univariate(f,x),lvar,lg,lval,leadlist,ldeg):List P
--JHD     (p0:P,p1:P):=(plist.first,plist.2)
--JHD     if univariate eval(p0,rest lvar,lval) ^= lg.first then
--JHD     (p0,p1):=(p1,p0)
--JHD     leadpol => p0
--JHD     cprim:=contprim([p0])
--JHD     cprim.first.prim
--JHD
--JHD -- Gcd for two multivariate polynomials
--JHD gcd(p1:P,p2:P) : P ==
--JHD   (p1:= abs(p1)) = (p2:= abs(p2)) => p1
--JHD   ground? p1 =>
--JHD   p1 = 1$P => p1
CHAPTER 8. CHAPTER G

--JHD p1 = 0$P => p2
--JHD ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
--JHD gcdTermList(p1,p2)
--JHD ground? p2 =>
--JHD p2 = 1$P => p2
--JHD p2 = 0$P => p1
--JHD gcdTermList(p2,p1)
--JHD mv1:= mainVariable(p1)::OV
--JHD mv2:= mainVariable(p2)::OV
--JHD mv1 = mv2 => gcdSameMainvar(p1,p2,mv1)
--JHD mv1 < mv2 => gcdTermList(p1,p2)
--JHD gcdTermList(p2,p1)
--JHD
--JHD -- Gcd for a list of multivariate polynomials
--JHD gcd(listp:List P) : P ==
--JHD lf:=sort(better,listp)
--JHD f:=lf.first
--JHD for g in lf.rest repeat
--JHD f:=gcd(f,g)
--JHD if f=1$P then return f
--JHD f
--JHD
--JHD -- Gcd and cofactors for a list of polynomials
--JHD gcdcofact(listp : List P) : List P ==
--JHD h:=gcd listp
--JHD cons(h,[(f exquo h) :: P for f in listp])
--JHD
--JHD -- Gcd for primitive polynomials
--JHD gcdprim(p1:P,p2:P):P ==
--JHD (p1:= abs(p1)) = (p2:= abs(p2)) => p1
--JHD ground? p1 =>
--JHD ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
--JHD p1 = 0$P => p2
--JHD 1$P
--JHD ground? p2 =>
--JHD p2 = 0$P => p1
--JHD 1$P
--JHD mv1:= mainVariable(p1)::OV
--JHD mv2:= mainVariable(p2)::OV
--JHD mv1 = mv2 =>
--JHD md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv1))
--JHD mp:=1$P
--JHD if md>1 then
--JHD mp:=(mv1::P)**md
--JHD p1:=(p1 exquo mp)::P
--JHD p2:=(p2 exquo mp)::P
--JHD mp*gcdPrim(p1,p2,mv1)
--JHD 1$P
--JHD
--JHD -- Gcd for a list of primitive multivariate polynomials
--JHD gcdprim(listp:List P) : P ==
package GENUPS GenerateUnivariatePowerSeries

-- GenerateUnivariatePowerSeries.input

)set break resume
)sys rm -f GenerateUnivariatePowerSeries.output
)spool GenerateUnivariatePowerSeries.output
)set message test on
(ev) set message auto off
(c) clear all

--S 1 of 1
(ev) show GenerateUnivariatePowerSeries
--E 1

)spool
)lisp (bye)

———
— GenerateUnivariatePowerSeries.help ——

====================================================================
GenerateUnivariatePowerSeries examples
====================================================================

GenerateUnivariatePowerSeries provides functions that create
power series from explicit formulas for their n-th coefficient.

See Also:
0 )show GenerateUnivariatePowerSeries

———

GenerateUnivariatePowerSeries (GENUPS)

Exports:
laurent puiseux series taylor

—— package GENUPS GenerateUnivariatePowerSeries ——
)abbrev package GENUPS GenerateUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 29 April 1990
++ Date Last Updated: 31 May 1990
++ Description:
++ \spadtype{GenerateUnivariatePowerSeries} provides functions that create
++ power series from explicit formulas for their \spad{n}th coefficient.

GenerateUnivariatePowerSeries(R,FE) : Exports == Implementation where
  R : Join(IntegralDomain,OrderedSet,RetractableTo Integer,  
    LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,  
    FunctionSpace R)
ANY1 ==> AnyFunctions1
EQ ==> Equation
I ==> Integer
NNI ==> NonNegativeInteger
RN ==> Fraction Integer
SEG ==> UniversalSegment
ST ==> Stream
SY ==> Symbol
UTS ==> UnivariateTaylorSeries
ULS ==> UnivariateLaurentSeries
UPXS ==> UnivariatePuiseuxSeries

Exports ==> with
  taylor: (I -> FE,EQ FE) -> Any
    ++ \spad{taylor(n +-> a(n),x = a)} returns \spad{sum(n = 0..,a(n)*(x-a)**n)}.
  taylor: (FE,SY,EQ FE) -> Any
    ++ \spad{taylor(a(n),n,x=a)} returns \spad{sum(n = 0..,a(n)*(x-a)**n)}.
  taylor: (I -> FE,EQ FE,SEG NNI) -> Any
    ++ \spad{taylor(n +-> a(n),x = a,n0..)} returns \spad{sum(n=n0..,a(n)*(x-a)**n)}.
    ++ \spad{taylor(n +-> a(n),x = a,n0..n1)} returns \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.
  laurent: (I -> FE,EQ FE,SEG I) -> Any
    ++ \spad{laurant(n +-> a(n),x = a)} returns \spad{sum(n = n0..,a(n) * (x - a)**n)}.
++ \spad{laurent(a(n),n,x=a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

puiseux: (RN -> FE,EQ FE,SEG RN,RN) -> Any
++ \spad{puiseux(n +-> a(n),x = a,r0..,r)} returns
++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{puiseux(n +-> a(n),x = a,r0..r1,r)} returns
++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

puiseux: (FE,SY,EQ FE,SEG RN,RN) -> Any
++ \spad{puiseux(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{puiseux(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

series: (I -> FE,EQ FE) -> Any
++ \spad{series(n +-> a(n),x = a)} returns
++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.

series: (FE,SY,EQ FE) -> Any
++ \spad{series(a(n),n,x = a)} returns
++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.

series: (I -> FE,EQ FE,SEG I) -> Any
++ \spad{series(n +-> a(n),x = a,n0..)} returns
++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
++ \spad{series(n +-> a(n),x = a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

series: (FE,SY,EQ FE,SEG I) -> Any
++ \spad{series(a(n),n,x=a,n0..)} returns
++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
++ \spad{series(a(n),n,x=a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

series: (RN -> FE,EQ FE,SEG RN,RN) -> Any
++ \spad{series(n +-> a(n),x = a,r0..,r)} returns
++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{series(n +-> a(n),x = a,r0..r1,r)} returns
++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

series: (FE,SY,EQ FE,SEG RN,RN) -> Any
++ \spad{series(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{series(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

Implementation ==> add

\makebox[\textwidth][c]{
\begin{verbatim}
Implementation ==> add

genStream: (I -> FE,I) -> ST FE
\makebox[\textwidth][c]{
genStream(f,n) == delay concat(f(n),genStream(f,n + 1))

genFiniteStream: (I -> FE,I,I) -> ST FE
\makebox[\textwidth][c]{
genFiniteStream(f,n,m) == delay
\makebox[\textwidth][c]{
  n > m => empty()
  concat(f(n),genFiniteStream(f,n + 1,m))
\end{verbatim}
}
taylor(f, eq) ==
  (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  coerce(series(genStream(f, 0))$UTS(FE, x, a))$ANY1(UTS(FE, x, a))

  taylor(an: FE, n: SY, eq: EQ FE) ==
    taylor((i:I): FE +-> eval(an, (n:: FE) = (i:: FE)), eq)

  taylor(f: I -> FE, eq: EQ FE, seg: SEG NNI) ==
    (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    hasHi seg =>
      n0 := lo seg; n1 := hi seg
      if n1 < n0 then (n0, n1) := (n1, n0)
      uts := series(genFiniteStream(f, n0, n1))$UTS(FE, x, a)
      uts := uts * monomial(1, n0)$UTS(FE, x, a)
      coerce(uts)$ANY1(UTS(FE, x, a))
    n0 := lo seg
    uts := series(genStream(f, n0))$UTS(FE, x, a)
    uts := uts * monomial(1, n0)$UTS(FE, x, a)
    coerce(uts)$ANY1(UTS(FE, x, a))

  taylor(an, n, eq, seg) ==
    taylor((i:I): FE +-> eval(an, (n:: FE) = (i:: FE)), eq, seg)

laurent(f, eq, seg) ==
  (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  hasHi seg =>
    n0 := lo seg; n1 := hi seg
    if n1 < n0 then (n0, n1) := (n1, n0)
    uts := series(genFiniteStream(f, n0, n1))$UTS(FE, x, a)
    uts := uts * monomial(1, n0)$ULS(FE, x, a)
    coerce(uts)$ANY1(ULS(FE, x, a))
    n0 := lo seg
    uts := series(genStream(f, n0))$UTS(FE, x, a)
    uts := uts * monomial(1, n0)$ULS(FE, x, a)
    coerce(uts)$ANY1(ULS(FE, x, a))

laurent(an, n, eq, seg) ==
  laurent((i:I): FE +-> eval(an, (n:: FE) = (i:: FE)), eq, seg)

modifyFcn:(RN -> FE, I, I, I) -> FE
modifyFcn(f, n0, nn, q, m) ==
  (zero?((m - n0) rem nn) => f(m/q); 0)

puiseux(f, eq, seg, r) ==
  (xx := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed" =>
    error "puiseux: left hand side must be a variable"
CHAPTER 8. CHAPTER G

\[ x := xx : SY; a := \text{rhs eq} \]

not positive? \( r \Rightarrow \text{error "puiseux: last argument must be positive"}

hasHi seg =>

\[ r0 := \text{lo seg}; r1 := \text{hi seg} \]

if \( r1 < r0 \) then \((r0,r1) := (r1,r0)\)

\[ p0 := \text{numer } r0; q0 := \text{denom } r0 \]

\[ p1 := \text{numer } r1; q1 := \text{denom } r1 \]

\[ p2 := \text{numer } r; q2 := \text{denom } r \]

\[ q := \text{lcm}(\text{lcm}(q0,q1),q2) \]

\[ n0 := p0 * (q \mod q0); n1 := p1 * (q \mod q1) \]

\[ nn := p2 * (q \mod q2) \]

ulsUnion :=

\[ \text{laurent}((i:I):\text{FE}+->\text{modifyFcn}(f,n0,nn,q,i),\text{eq},\text{segment}(n0,n1)) \]

uls := retract(ulsUnion)$\text{ANY1}(\text{ULS}(\text{FE},x,a))$

coerce(puiseux(1/q,uls)$\text{UPXS}(\text{FE},x,a))$\text{ANY1}(\text{UPXS}(\text{FE},x,a))

puiseux(an,n,eq,r0,m) ==

\[ \text{puiseux}((r:RN):\text{FE}+->\text{eval}(an,(n::\text{FE}) = (r::\text{FE})),\text{eq},r0,m) \]

series(f:I -> \text{FE},\text{eq}:\text{EQ FE}) == puiseux(r+->f(\text{numer } r),\text{eq},\text{segment } 0,1)

series(an:\text{FE},n:SY,\text{eq}:\text{EQ FE}) == puiseux(an,n,eq,\text{segment } 0,1)

series(f:I -> \text{FE},\text{eq}:\text{EQ FE},\text{seg}:\text{SEG I}) ==

\[ \text{ratSeg} : \text{SEG RN} := \text{map}(x+->x::\text{RN},\text{seg})$\text{UniversalSegmentFunctions2}(I,\text{RN}) \]

puiseux((r:RN):\text{FE}+->f(\text{numer } r),\text{eq},\text{ratSeg},1)

series(an:\text{FE},n:SY,\text{eq}:\text{EQ FE},\text{seg}:\text{SEG I}) ==

\[ \text{ratSeg} : \text{SEG RN} := \text{map}(i+->i::\text{RN},\text{seg})$\text{UniversalSegmentFunctions2}(I,\text{RN}) \]

puiseux(an,n,eq,\text{ratSeg},1)

series(f:RN -> \text{FE},\text{eq}:\text{EQ FE},\text{seg}:\text{SEG RN},r:RN) == puiseux(f,\text{eq},\text{seg},r)

series(an:\text{FE},n:SY,\text{eq}:\text{EQ FE},\text{seg}:\text{SEG RN},r:RN) == puiseux(an,n,eq,\text{seg},r)

— GENUPS.dotabb —
package GENEEZ GenExEuclid

---

<table>
<thead>
<tr>
<th>GenExEuclid.input</th>
</tr>
</thead>
</table>

)set break resume
)sys rm -f GenExEuclid.output
)spool GenExEuclid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GenExEuclid
--E 1

)spool
)lisp (bye)

---

<table>
<thead>
<tr>
<th>GenExEuclid.help</th>
</tr>
</thead>
</table>

====================================================================
GenExEuclid examples
====================================================================

The equation Af+Bg=h and its generalization to n polynomials
is solved for solutions over the R, euclidean domain.

A table containing the solutions of Af+Bg=x**k is used.

The operations are performed modulus a prime which are in principle
big enough, but the solutions are tested and, in case of failure, a
hensel lifting process is used to get to the right solutions. It will
be used in the factorization of multivariate polynomials over finite
field, with R=F[x].

See Also:
o )show GenExEuclid

---
GenExEuclid (GENEEZ)

Exports:
compBound  reduction  solveid  tablePow  testModulus

— package GENEEZ GenExEuclid —

)abbrev package GENEEZ GenExEuclid
++ Author : P.Gianni.
++ Date Created: January 1990
++ Description:

GenExEuclid(R,BP) : C == T
where
  R : EuclideanDomain
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  BP : UnivariatePolynomialCategory R
  L ==> List
C == with
  reduction: (BP,R) -> BP
    ++ reduction(p,prime) reduces the polynomial p modulo prime of R.
    ++ Note that this function is exported only because it’s conditional.
  compBound: (BP,L BP) -> NNI
    ++ compBound(p,lp)
    ++ computes a bound for the coefficients of the solution
    ++ polynomials.
    ++ Given a polynomial right hand side p,
    ++ and a list lp of left hand side polynomials.
    ++ Exported because it depends on the valuation.
  tablePow : (NNI,R,L BP) -> Union(Vector(L BP),"failed")
    ++ tablePow(maxdeg,prime,lpol) constructs the table with the
    ++ coefficients of the Extended Euclidean Algorithm for lpol.
    ++ Here the right side is \spad{x**k}, for k less or equal to maxdeg.
    ++ The operation returns "failed" when the elements
    ++ are not coprime modulo prime.
solveid : (BP,R,Vector L BP) -> Union(L BP,"failed")
  ++ solveid(h,table) computes the coefficients of the
  ++ extended euclidean algorithm for a list of polynomials
  ++ whose tablePow is table and with right side h.

  testModulus : (R, L BP) -> Boolean
  ++ testModulus(p,lp) returns true if the the prime p
  ++ is valid for the list of polynomials lp, i.e. preserves
  ++ the degree and they remain relatively prime.

T == add
  if R has multiplicativeValuation then
    compBound(m:BP,listpolys:L BP) : NNI ==
      ldeg:= [degree f for f in listpolys]
      n:NNI:= (+/[df for df in ldeg])
      normlist:=[ +/[euclideanSize(u)**2 for u in coefficients f] for f in listpolys]
      nm:= +/[euclideanSize(u)**2 for u in coefficients m] for f in listpolys])
      normprod := */ [g**((n-df)::NNI) for g in normlist for df in ldeg]
      2*(approxSqrt(normprod * nm)$IntegerRoots(Integer))::NNI
  else if R has additiveValuation then
    -- a fairly crude Hadamard-style bound for the solution
    -- based on regarding the problem as a system of linear equations.
    compBound(m:BP,listpolys:L BP) : NNI ==
      "max"/[euclideanSize u for u in coefficients m] +
      +/["max"/[euclideanSize u for u in coefficients p] for p in listpolys]
  else
    compBound(m:BP,listpolys:L BP) : NNI ==
      error "attempt to use compBound without a well-understood valuation"

  if R has IntegerNumberSystem then
    reduction(u:BP,p:R):BP ==
      p = 0 => u
      map(x +-> symmetricRemainder(x,p),u)
  else reduction(u:BP,p:R):BP ==
      p = 0 => u
      map(x +-> x rem p,u)

merge(p:R,q:R):Union(R,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

modInverse(c:R,p:R):R ==
  (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

  invlcv:=modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlcv*v,p))
reduction(r.remainder,p) ^=0 => "failed"
reduction(invlcr*r.quotient,p)

FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)

--make table global variable!
table:Vector L BP
import GeneralHenselPackage(R,BP)

--local functions
makeProducts : L BP -> L BP
liftSol: (L BP,BP,R,R,Vector L BP,BP,NNI) -> Union(L BP,"failed")
reduceList(lp:L BP,lmod:R): L FP == [reduce(ff,lmod) for ff in lp]
coerceLFP(lf:L FP):L BP == [fm::BP for fm in lf]
euclideanSize(lmodk) > bound => "failed"
d:=degree err
ftab:Vector L FP :=
   map(x +-> reduceList(x,lmod),table)$VectorFunctions2(List BP,List FP)
sln:L FP:=[$0$FP for xx in ftab.1 ]
for i in 0 .. d |((cc:=coefficient(err,i)) ^=0 repeat
   sln:=[slp+reduce(cc::BP,lmod)*pp
       for pp in ftab.(i+1) for slp in sln]
nsol:=[f-lmodk*reduction(g::BP,lmod) for f in oldsol for g in sln]
lmodk1:=lmod*lmodk
nsol:=[reduction(slp,lmodk1) for slp in nsol]
lpolys:L BP:=table.(#table)
   (fs:=-/[f*g for f in lpolys for g in nsol]) = m => nsol
a:BP:=((fs-m) exquo lmodk1)::BP
liftSol(nsol,a,lmod,lmodk1,table,m,bound)

makeProducts(listPol:L BP):L BP ==
   #listPol < 2 => listPol
   #listPol = 2 => reverse listPol
   f:= first listPol
   ll := rest listPol
   [*ll,[:f*g for g in makeProducts ll]]

testModulus(pmod, listPol) ==
   redListPol := reduceList(listPol, pmod)
   for pol in listPol for rpol in redListPol repeat
      degree(pol) ^= degree(rpol::BP) => return false
   while not empty? redListPol repeat
      rpol := first redListPol
      redListPol := rest redListPol
      for rpol2 in redListPol repeat

\[
gcd(rpol, rpol2) \not= 1 \Rightarrow \text{return false}
true
\]

if R has Field then
\[
tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
multiE:=multiEuclidean(listPol,1$BP)
multiE case "failed" => "failed"
ptable:Vector L BP :=new(mdeg+1,[]) ptable.1:=multiE
x:BP:=monomial(1,1)
for i in 2..mdeg repeat ptable.i:=
    [tpol*x rem fpol for tpol in ptable.(i-1) for fpol in listPol]
ptable.(mdeg+1):=makeProducts listPol
ptable
\]
solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
    -- Actually, there's no possibility of failure
d:=degree m
sln:L BP:=[0$BP for xx in table.1]
for i in 0 .. d | coefficient(m,i)=0 repeat
    sln:=[slp+coefficient(m,i)*pp
    for pp in table.(i+1) for slp in sln]
sln
else
\[
tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
listP:L FP:= [reduce(pol,pmod) for pol in listPol]
multiE:=multiEuclidean(listP,1$FP)
multiE case "failed" => "failed"
ftable:Vector L FP :=new(mdeg+1,[])
fl:L FP:= [ff::FP for ff in multiE]
ftable.1:=[fpol for fpol in fl]
x:FP:=reduce(monomial(1,1),pmod)
for i in 2..mdeg repeat ftable.i:=
    [tpol*x rem fpol for tpol in ftable.(i-1) for fpol in listP]
ptable:= map(coerceLFP,ftable)$VectorFunctions2(List FP,List BP)
ptable.(mdeg+1):=makeProducts listPol
ptable
\]
solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
d:=degree m
ftab:Vector L FP:=
    map(x+->reduceList(x,pmod),table)$VectorFunctions2(List BP,List FP)
lpolys:L BP:=table.(#table)
sln:L FP:=[0$FP for xx in ftab.1]
for i in 0 .. d | coefficient(m,i)=0 repeat
    sln:=[slp+reduce(coefficient(m,i)::BP,pmod)*pp
    for pp in ftab.(i+1) for slp in sln]
soln:=[slp::BP for slp in sln]
(fs:+[f*g for f in lpolys for g in soln]) = m=> soln
-- Compute bound
bound:=compBound(m,lpolys)
a:BP:=((fs-m) exquo pmod)::BP
liftSol(soln,a,pmod,pmod,table,m,bound)

---

— GENEEZ.dotabb —

"GENEEZ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENEEZ"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GENEEZ" -> "PFECAT"

---

package GENUFACT GenUFactorize

— GenUFactorize.input —

)set break resume
)sys rm -f GenUFactorize.output
)spool GenUFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GenUFactorize
--E 1

)spool
)lisp (bye)

---

— GenUFactorize.help —

====================================================================
GenUFactorize examples
====================================================================

This package provides operations for the factorization of univariate
polynomials with integer coefficients. The factorization is done by "lifting" the finite "berlekamp’s" factorization

See Also:
  o )show GenUFactorize

---

GenUFactorize (GENUFAC'T)

Exports:
  factor

--- package GENUFACT GenUFactorize ---

)abbrev package GENUFACT GenUFactorize
++ Description:
  ++ This package provides operations for the factorization
  ++ of univariate polynomials with integer
  ++ coefficients. The factorization is done by "lifting" the
  ++ finite "berlekamp’s" factorization

GenUFactorize(R) : public == private where
  R : EuclideanDomain
  PR ==> SparseUnivariatePolynomial R -- with factor
    -- should be UnivariatePolynomialCategory
  NNI ==> NonNegativeInteger
  SUP ==> SparseUnivariatePolynomial

  public == with
    factor : PR -> Factored PR
      ++ factor(p) returns the factorisation of p
private == add

 Factorisation currently fails when algebraic extensions have multiple
generators.

\[
\text{factorWarning}(f:\text{OutputForm}):\text{Void} ==
\]
\[
\text{import AnyFunctions1(String)}
\]
\[
\text{import AnyFunctions1(OutputForm)}
\]
\[
\text{outputList(}[[\text"WARNING (genufact): No known algorithm to factor "}, _
f::\text{Any, _
"}, \text" trying square-free."}],\text{Any}\})$\text{OutputPackage}
\]

\[
\text{factor}(f:PR) : \text{Factored PR} ==
\]
\[
\text{R is Integer} \Rightarrow (\text{factor } f)$\text{GaloisGroupFactorizer}(PR)
\]
\[
\text{R is Fraction Integer} \Rightarrow (\text{factor } f)$\text{RationalFactorize}(PR)
\]
\[
\text{R has Field and R has Finite} \Rightarrow
\]
\[
\text{R has FiniteFieldCategory} \Rightarrow (\text{factor } f)$\text{DistinctDegreeFactorize}(R,PR)
\]
\[
\text{R is (Complex Integer)} \Rightarrow (\text{factor } f)$\text{ComplexFactorization}(Integer,PR)
\]
\[
\text{R is (Complex Fraction Integer)} \Rightarrow (\text{factor } f)$\text{ComplexFactorization(Fraction Integer,PR)}
\]
\[
\text{R is AlgebraicNumber} \Rightarrow (\text{factor } f)$\text{AlgFactor}(PR)
\]

-- following is to handle SAE

\[
\text{R has generator : ()} \Rightarrow \text{R} \Rightarrow
\]
\[
\text{var} := \text{symbol(convert(generator())::OutputForm}@\text{InputForm})
\]
\[
\text{up}:=\text{UnivariatePolynomial(var,Fraction Integer)}
\]
\[
\text{R has MonogenicAlgebra(Fraction Integer, up)} \Rightarrow
\]
\[
\text{factor}(f)$\text{SimpleAlgebraicExtensionAlgFactor}(up, R, PR)
\]
\[
\text{uppp}:=\text{UnivariatePolynomial(var,Fraction Polynomial Integer)}
\]
\[
\text{R has MonogenicAlgebra(Fraction Polynomial Integer, uppp)} \Rightarrow
\]
\[
\text{factor}(f)$\text{SAERationalFunctionAlgFactor}(uppp, R, PR)
\]
\[
\text{factorWarning}(f::\text{OutputForm})
\]
\[
\text{squareFree } f
\]
\[
\text{factorWarning}(f::\text{OutputForm})
\]
\[
\text{squareFree } f
\]

—— GENUFACT.dotabb ——

"GENUFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENUFACT"]
package INTG0 GenusZeroIntegration

<table>
<thead>
<tr>
<th>GenusZeroIntegration.input</th>
</tr>
</thead>
<tbody>
<tr>
<td>)set break resume</td>
</tr>
<tr>
<td>)sys rm -f GenusZeroIntegration.output</td>
</tr>
<tr>
<td>)spool GenusZeroIntegration.output</td>
</tr>
<tr>
<td>)set message test on</td>
</tr>
<tr>
<td>)set message auto off</td>
</tr>
<tr>
<td>)clear all</td>
</tr>
</tbody>
</table>

--S 1 of 1
)show GenusZeroIntegration
--E 1

)spool
)lisp (bye)

---

<table>
<thead>
<tr>
<th>GenusZeroIntegration.help</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenusZeroIntegration examples</td>
</tr>
</tbody>
</table>

Rationalization of several types of genus 0 integrands;
This internal package rationalises integrands on curves of the form:

\[ y^2 = a x^2 + b x + c \]
\[ y^2 = (a x + b) / (c x + d) \]
\[ f(x, y) = 0 \] where \( f \) has degree 1 in \( x \)

The rationalization is done for integration, limited integration,
extended integration and the risch differential equation;

See Also:
- )show GenusZeroIntegration
GenusZeroIntegration (INTG0)

Exports:
- lift
- multivariate
- palgLODE0
- palgRDE0
- palgextint0
- palgint0
- palglimint0
- univariate

--- package INTG0 GenusZeroIntegration ---

\texttt{GenusZeroIntegration(R, F, L): Exports == Implementation where}
\begin{align*}
R & : \text{Join(GcdDomain, RetractableTo Integer, OrderedSet, CharacteristicZero, LinearlyExplicitRingOver Integer)} \\
F & : \text{Join(FunctionSpace R, AlgebraicallyClosedField, TranscendentalFunctionCategory)} \\
L & : \text{SetCategory}
\end{align*}
\begin{align*}
\text{SY} & \Rightarrow \text{Symbol} \\
\text{Q} & \Rightarrow \text{Fraction Integer} \\
\text{K} & \Rightarrow \text{Kernel F} \\
\text{P} & \Rightarrow \text{SparseMultivariatePolynomial(R, K)} \\
\text{UP} & \Rightarrow \text{SparseUnivariatePolynomial F}
\end{align*}
RF ==> Fraction UP
UPUP ==> SparseUnivariatePolynomial RF
IR ==> IntegrationResult F
LOG ==> Record(coeff:F, logand:F)
U1 ==> Union(F, "failed")
U2 ==> Union(Record(ratpart:F, coeff:F),"failed")
U3 ==> Union(Record(mainpart:F, limitedlogs:List LOG), "failed")
REC ==> Record(coeff:F, var:List K, val:List F)
ODE ==> Record(particular: Union(F, "failed"), basis: List F)
LDO0==> LinearOrdinaryDifferentialOperator1 RF

Exports ==> with
palgint0 : (F, K, K, F, UP) -> IR
++ palgint0(f, x, y, d, p) returns the integral of \( f(x,y)dx \)
++ where \( y \) is an algebraic function of \( x \) satisfying
++ \( spad{d(x)^2 y(x)^2 = P(x)} \).
palgint0 : (F, K, K, K, F, RF) -> IR
++ palgint0(f, x, y, z, t, c) returns the integral of \( f(x,y)dx \)
++ where \( y \) is an algebraic function of \( x \) satisfying
++ \( spad{f(x,y)dx = c f(t,y) dy}; c \) and \( t \) are rational functions of \( y \).  
++ Argument \( z \) is a dummy variable not appearing in \( spad{f(x,y)} \).

palgextint0: (F, K, K, K, F, F, UP) -> U2
++ palgextint0(f, x, y, g, d, p) returns functions \( \{h, c\} \) such
++ that \( spad{dh/dx = f(x,y) - c g}, \) where \( y \) is an algebraic function
++ of \( x \) satisfying \( spad{d(x)^2 y(x)^2 = P(x)} \),
++ or "failed" if no such functions exist.

palgextint0: (F, K, K, K, F, K, F, RF) -> U2
++ palgextint0(f, x, y, g, z, t, c) returns functions \( \{h, d\} \) such
++ that \( spad{dh/dx = f(x,y) - d g}, \) where \( y \) is an algebraic function
++ of \( x \) satisfying \( spad{f(x,y)dx = c f(t,y) dy}; c \) and \( t \) are
++ rational functions of \( y \).
++ Argument \( z \) is a dummy variable not appearing in \( spad{f(x,y)} \).
++ The operation returns "failed" if no such functions exist.

palglimint0: (F, K, K, List F, F, UP) -> U3
++ palglimint0(f, x, y, \[u1,...,un\], d, p) returns functions \( \{h, \{ci, ui\}\} \)
++ such that \( spad{h + \sum(ci \log(ui))/dx = f(x,y)} \) if such functions exist,
++ and "failed" otherwise.
++ Argument \( y \) is an algebraic function of \( x \) satisfying
++ \( spad{d(x)^2 y(x)^2 = P(x)} \).

palglimint0: (F, K, K, List F, K, F, RF) -> U3
++ palglimint0(f, x, y, \[u1,...,un\], z, t, c) returns functions
++ \( spad{\{h, \{ci, ui\}\}} \) such that the \( ui \)'s are among \( spad{\{u1,...,un\}} \)
++ and \( spad{d(h + \sum(ci \log(ui))/dx = f(x,y)} \) if such functions exist,
++ and "failed" otherwise.
++ Argument \( y \) is an algebraic function of \( x \) satisfying
++ \( spad{f(x,y)dx = c f(t,y) dy}; c \) and \( t \) are rational functions of \( y \).

palgRDE0 : (F, F, K, K, (F, F, SY) -> U1, F, UP) -> U1
++ palgRDE0(f, g, x, y, foo, d, p) returns a function \( spad{z(x,y)} \)
++ such that \( spad{dz/dx + n * df/dx z(x,y) = g(x,y)} \) if such a \( z \) exists,
++ and "failed" otherwise.
++ Argument y is an algebraic function of x satisfying
++ \( \text{spad}(d(x)^2y(x)^2 = P(x)) \).
++ Argument foo, called by \text{spad}(\text{foo}(a, b, x)), is a function that solves
++ \( \text{spad}(du/dx + n * da/dx u(x) = u(x)) \)
++ for an unknown \( \text{spad}(u(x)) \) not involving y.

\text{palgRDE0} : (F, F, K, K, (F, F, SY) -> U1, K, F, RF) -> U1
++ \text{palgRDE0}(f, g, x, y, foo, t, c) returns a function \( \text{spad}(z(x,y)) \)
++ such that \( \text{spad}(dz/dx + n * df/dx z(x,y) = g(x,y)) \) if such a z exists,
++ and "failed" otherwise.
++ Argument y is an algebraic function of x satisfying
++ \( \text{spad}(f(x,y)dx = c f(t,y) dy) \); c and t are rational functions of y.
++ Argument \( \text{spad}(\text{foo}) \), called by \( \text{spad}(\text{foo}(a, b, x)) \), is a function that
++ solves \( \text{spad}(du/dx + n * da/dx u(x) = u(x)) \)
++ for an unknown \( \text{spad}(u(x)) \) not involving y.

\text{univariate} : (F, K, K, UP) -> UP
++ \text{univariate}(f,k,k,p) \text{ undocumented}

\text{multivariate} : (UPUP, K, F) -> F
++ \text{multivariate}(u,k,f) \text{ undocumented}

\text{lift} : (UP, K) -> UP
++ \text{lift}(u,k) \text{ undocumented}

If L has LinearOrdinaryDifferentialOperatorCategory F then
\text{palgLODE0} : (L, F, K, K, F, UP) -> ODE
++ \text{palgLODE0}(op, g, x, y, d, p) returns the solution of \( \text{spad}(op f = g) \).
++ Argument y is an algebraic function of x satisfying
++ \( \text{spad}(d(x)^2y(x)^2 = P(x)) \).

\text{palgLODE0} : (L, F, K, K, K, F, RF) -> ODE
++ \text{palgLODE0}(op,g,x,y,z,t,c) returns the solution of \( \text{spad}(op f = g) \)
++ Argument y is an algebraic function of x satisfying
++ \( \text{spad}(f(x,y)dx = c f(t,y) dy) \); c and t are rational functions of y.

\text{Implementation} ==>
\text{add}
import RationalIntegration(F, UP)
import AlgebraicManipulations(R, F)
import IntegrationResultFunctions2(RF, F)
import ElementaryFunctionStructurePackage(R, F)
import SparseUnivariatePolynomialFunctions2(F, RF)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

\text{mkRat} : (F, REC, List K) -> RF
\text{mkRatlx} : (F, K, K, F, K, RF) -> RF
\text{quads subst}: (K, K, F, UP) -> Record(diff:F, subs:REC, newk:List K)
\text{kerdiff} : (F, F) -> List K
\text{checkroot}: (F, List K) -> F
\text{univ} : (F, List K, K) -> RF

\text{dummy} := \text{kernel}(\text{new}()$SY)@K

\text{kerdiff}(sa, a) == \text{setDifference}($\text{kernels}$ sa, kernels a)
checkroot(f, l) == (empty? l => f; rootNormalize(f, first l))
univ(c, l, x) == univariate(checkroot(c, l), x)
univariate(f, x, y, p) == lift(univariate(f, y, p), x)
lift(p, k) == map(x1+->univariate(x1, k), p)

palgint0(f, x, y, den, radi) ==
  -- y is a square root so write f as f1 y + f0 and integrate separately
  ff := univariate(f, x, y, minPoly y)
  f0 := reductum ff
  pr := quadsubst(x, y, den, radi)
  map(f1+->f1(x::F), integrate(retract(f0)@RF)) +
  map(f1+->f1(pr.diff), integrate
  mkRat(multivariate(leadingMonomial ff,x,y::F), pr.subs, pr.newk))

-- the algebraic relation is (den * y)**2 = p where p is a * x**2 + b * x + c
-- if p is squarefree, then parametrize in the following form:
  -- u = y - x \sqrt{a}
  -- x = (u^2 - c) / (b - 2 u \sqrt{a}) = h(u)
  -- dx = h'(u) du
  -- y = (u + a h(u)) / den = g(u)
-- if a is a perfect square,
  -- u = (y - \sqrt{c}) / x
  -- x = (b - 2 u \sqrt{c}) / (u^2 - a) = h(u)
  -- dx = h'(u) du
  -- y = (u h(u) + \sqrt{c}) / den = g(u)
-- otherwise.
-- if p is a square p = a t^2, then we choose only one branch for now:
  -- u = x
  -- x = u = h(u)
  -- dx = du
  -- y = t \sqrt{a} / den = g(u)
-- returns [u(x,y), [h'(u), [x, y], [h(u), g(u)], l] in both cases,
-- where l is empty if no new square root was needed,
-- l := [k] if k is the new square root kernel that was created.
quadsubst(x, y, den, p) ==
  u := dummy::F
  b := coefficient(p, 1)
  c := coefficient(p, 0)
  sa := rootSimp sqrt(a := coefficient(p, 2))
  zero?(b * b - 4 * a * c) => -- case where p = a (x + b/(2a))^2
    [x::F, [1, [x, y], [u, sa * (u + b / (2*a)) / eval(den,x,u)]], empty()]
empty? kerdiff(sa, a) =>
  bm2u := b - 2 * u * sa
  q := eval(den, x, xx := (u**2 - c) / bm2u)
  yy := (ua := u + xx * sa) / q
  [y::F - x::F * sa, [2 * ua / bm2u, [x, y], [xx, yy]], empty()]
  u2ma := u**2 - a
  sc := rootSimp sqrt c
  q := eval(den, x, xx := (b - 2 * u * sc) / u2ma)
\[
yy := (ux := xx * u + sc) / q
\]

\[
[(y::F - sc) / x::F, [- 2 * ux / u2ma, [x ,y], [xx, yy]], \text{kerdiff}(sc, c)]
\]

\[
mkRatlx(f,x,y,t,z,dx) ==
\]
\[
\text{rat} := \text{univariate}(\text{eval}(f, [x, y], [t, z::F]), z) * dx
\]
\[
\text{numer}(\text{rat}) / \text{denom}(\text{rat})
\]

\[
mkRat(f, \text{rec}, l) ==
\]
\[
\text{rat} := \text{univariate}(\text{checkroot}(\text{rec}.\text{coeff} * \text{eval}(f,\text{rec}.\text{var},\text{rec}.\text{val}), l), \text{dummy})
\]
\[
\text{numer}(\text{rat}) / \text{denom}(\text{rat})
\]

\[
\text{palgint0}(f, x, y, z, xx, dx) ==
\]
\[
\text{map}(x1\rightarrow\text{multivariate}(x1, y), \text{integrate}\ mkRatlx(f, x, y, xx, z, dx))
\]

\[
\text{palgestint0}(f, x, y, g, z, xx, dx) ==
\]
\[
\text{map}(x1\rightarrow\text{multivariate}(x1, y), \text{extendedint}(mkRatlx(f,x,y,xx,z,dx), mkRatlx(g,x,y,xx,z,dx)))
\]

\[
\text{palglimint0}(f, x, y, lu, z, xx, dx) ==
\]
\[
\text{map}(x1\rightarrow\text{multivariate}(x1, y), \text{limitedint}(mkRatlx(f, x, y, xx, z, dx), [\text{mkRatlx}(u, x, y, xx, z, dx) \text{ for } u \text{ in } lu])
\]

\[
\text{palgRDE0}(f, g, x, y, \text{rischde}, z, xx, dx) ==
\]
\[
(u := \text{rischde}(\text{eval}(f, [x, y], [xx, z::F]), \text{multivariate}(dx, z) * \text{eval}(g, [x, y], [xx, z::F]), \text{symbolIfCan}(z::SY)) \text{ case "failed" => "failed"})
\]
\[
\text{eval}(u::F, z, y::F)
\]

-- given \( p = \sum_i a_i(x) Y^i \), returns \( \sum_i a_i(x) y^i \)
\[
\text{multivariate}(p, x, y) ==
\]
\[
(\text{map}((x1:RF):F\rightarrow\text{multivariate}(x1, x), p)$\text{SparseUnivariatePolynomialFunctions2}(RF, F))
\]
\[
(y)
\]

\[
\text{palgestint0}(f, x, y, g, \text{den}, \text{radi}) ==
\]
\[
\text{pr} := \text{quadsubst}(x, y, \text{den}, \text{radi})
\]
\[
\text{map}(f1\rightarrow f1(\text{pr}.\text{diff}),
\text{extendedint}(\text{mkRat}(f, \text{pr}.\text{subs}, \text{pr}.\text{newk}), \text{mkRat}(g, \text{pr}.\text{subs}, \text{pr}.\text{newk})))
\]

\[
\text{palglimint0}(f, x, y, \text{lu}, \text{den}, \text{radi}) ==
\]
\[
\text{pr} := \text{quadsubst}(x, y, \text{den}, \text{radi})
\]
\[
\text{map}(f1\rightarrow f1(\text{pr}.\text{diff}),
\text{limitedint}(\text{mkRat}(f, \text{pr}.\text{subs}, \text{pr}.\text{newk}),
[\text{mkRat}(u, \text{pr}.\text{subs}, \text{pr}.\text{newk}) \text{ for } u \text{ in } lu]))
\]

\[
\text{palgRDE0}(f, g, x, y, \text{rischde}, \text{den}, \text{radi}) ==
\]
\[
\text{pr} := \text{quadsubst}(x, y, \text{den}, \text{radi})
\]
\[
(u := \text{rischde}((\text{checkroot}(\text{eval}(f, \text{pr}.\text{subs}.\text{var}, \text{pr}.\text{subs}.\text{val}), \text{pr}.\text{newk}),
\text{checkroot}(\text{pr}.\text{subs}.\text{coeff} * \text{eval}(g, \text{pr}.\text{subs}.\text{var}, \text{pr}.\text{subs}.\text{val}),
\text{pr}.\text{newk}), \text{symbolIfCan}(\text{dummy}::SY)) \text{ case "failed"})
eval(u::F, dummy, pr.diff)

if L has LinearOrdinaryDifferentialOperatorCategory F then
  import RationalLODE(F, UP)

palgLODE0(eq, g, x, y, den, radi) ==
  pr := quadsubst(x, y, den, radi)
  d := monomial(univ(inv(pr.subs.coef), pr.newk, dummy), 1)$LODO
  di:LODO := 1  -- will accumulate the powers of d
  op:LODO := 0  -- will accumulate the new LODO
  for i in 0..degree eq repeat
    op := op + univ(eval(coefficient(eq, i), pr.subs.var, pr.subs.val),
        pr.newk, dummy) * di
    di := d * di
  rec := ratDsolve(op, univ(eval(g, pr.subs.var, pr.subs.val), pr.newk, dummy))
  bas:List(F) := [b(pr.diff) for b in rec.basis]
  rec.particular case "failed" => ["failed", bas]
  [((rec.particular)::RF) (pr.diff), bas]

palgLODE0(eq, g, x, y, kz, xx, dx) ==
  d := monomial(univariate(inv multivariate(dx, kz), kz), 1)$LODO
  di:LODO := 1  -- will accumulate the powers of d
  op:LODO := 0  -- will accumulate the new LODO
  lk:List(K) := [x, y]
  lv:List(F) := [xx, kz::F]
  for i in 0..degree eq repeat
    op := op + univariate(eval(coefficient(eq, i), lk, lv), kz) * di
    di := d * di
  rec := ratDsolve(op, univariate(eval(g, lk, lv), kz))
  bas:List(F) := [multivariate(b, y) for b in rec.basis]
  rec.particular case "failed" => ["failed", bas]
  [multivariate((rec.particular)::RF, y), bas]

INTG0.dotabb

"INTG0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTG0"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"INTG0" -> "FS"
"INTG0" -> "ACF"
package GDRAW GnuDraw

--- GnuDraw.input ---

)set break resume
)sys rm -f GnuDraw.output
)spool GnuDraw.output
)set message test on
)set message auto off
)clear all
--) 1 of 1
)show GnuDraw
--R
--R GnuDraw is a package constructor
--R Abbreviation for GnuDraw is GDRAW
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GDRAW
--R
--R---------------------------------- Operations ----------------------------------
--R gnudraw : (Expression(Float),SegmentBinding(Float),String,List(DrawOption)) -> Void
--R gnudraw : (Expression(Float),SegmentBinding(Float),String) -> Void
--R gnudraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String,List(DrawOption)) -> Void
--R gnudraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- GnuDraw.help ---

====================================================================
GnuDraw examples
====================================================================

This package provides support for gnuplot. These routines generate output files contain gnuplot scripts that may be processed directly by gnuplot. This is especially convenient in the axiom-wiki environment where gnuplot is called from LaTeX via gnuplottex.

See Also:
  o )show GnuDraw
GnuDraw (GDRAW)

--- package GDRAW GnuDraw ---

)abbrev package GDRAW GnuDraw
++ Author: Bill Page and David Cyganski
++ Date: June 25, 2008
++ Description:
++ This package provides support for gnuplot. These routines
++ generate output files contain gnuplot scripts that may be
++ processed directly by gnuplot. This is especially convenient
++ in the axiom-wiki environment where gnuplot is called from
++ LaTeX via gnuplottex.
GnuDraw(): Exports == Implementation where

EF ==> Expression Float
SBF ==> SegmentBinding Float
DROP ==> DrawOption
DROP0 ==> DrawOptionFunctions0
STR ==> String
Exports ==> with

gnuDraw:(EF, SBF, STR, List DROP)->Void
  ++ \spad{gnuDraw} provides 2d plotting with options
  ++
  ++X gnuDraw(D(cos(exp(z))/exp(z^2),z),z=-5..5,"out2d.dat",title="out2d")
  ++X )sys gnuplot -persist out2d.dat

gnuDraw:(EF, SBF, STR)->Void
  ++ \spad{gnuDraw} provides 2d plotting, default options
++ \spad{gnuDraw} provides 3d surface plotting with options
++
++ \spad{gnuDraw(sin(x)*cos(y),x=-6..4,y=-4..6,"out3d.dat",title=="out3d")}
++ \sys gnuplot -persist out3d.dat

Implementations \=> add
-- 2-d plotting
\begin{verbatim}
import TwoDimensionalViewport, GraphImage, TopLevelDrawFunctions EF
f1:TextFile:=open(filename::FileName,"output")
-- handle optional parameters
writeLine!(f1,concat(["set title ",title(opts,"")$DROP0,"""]))
writeLine!(f1,"plot '-' title '' lw 3 with lines")
-- extract data as List List Point DoubleFloat
p2:=pointLists(getGraph(draw(f, segbind),1));
for p1 in p2 repeat
  for p in p1 repeat
    writeLine!(f1,concat([unparse(convert(p.1)@InputForm)," ",
        unparse(convert(p.2)@InputForm)])
writeLine!(f1); -- blank line need to mark a "branch"
close! f1
\end{verbatim}
-- default title is ""
\begin{verbatim}
gnuDraw(f,segbind,filename,[title("")$DROP])
\end{verbatim}
-- 3-d plotting
\begin{verbatim}
import SubSpace, ThreeSpace DoubleFloat, TopLevelDrawFunctions EF
f1:TextFile:=open(filename::FileName,"output")
-- process optional parameters
writeLine!(f1,concat(["set title ",title(opts,"")$DROP0,"""]))
writeLine!(f1,"splot '-' title '' with pm3d")
-- extract data as List List Point DoubleFloat
p2:=mesh(subspace(draw(f, segbind1, segbind2)));
for p1 in p2 repeat
  for p in p1 repeat
    writeLine!(f1,concat([unparse(convert(p.1)@InputForm)," ",
        unparse(convert(p.2)@InputForm)," ",
        unparse(convert(p.3)@InputForm)])
writeLine!(f1); -- blank line need to mark a "branch"
close! f1
\end{verbatim}
unparse(convert(p.3@InputForm)))
   writeLine!(f1); -- blank line need to mark a "branch"
close! f1

-- default title is ""
gnuDraw(f,segbind1,segbind2,filename,[title("")$DROP])

---

<table>
<thead>
<tr>
<th>GDRAW.dotabb</th>
</tr>
</thead>
</table>
"GDRAW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GDRAW"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"GDRAW" -> "FS"

---

package GOSPER GosperSummationMethod

--- GosperSummationMethod.input ---

)set break resume
)sys rm -f GosperSummationMethod.output
)spool GosperSummationMethod.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GosperSummationMethod
--E 1

)spool
)lisp (bye)

---

--- GosperSummationMethod.help ---

====================================================================
GosperSummationMethod examples
====================================================================
Gosper's summation algorithm.

See Also:
- GosperSummationMethod

GosperSummationMethod (GOSPER)

Exports:
GosperSummationMethod

— package GOSPER GosperSummationMethod —

)abbrev package GOSPER GosperSummationMethod
++ Author: SMW
++ Date Last Updated: 19 August 1991
++ Description:
++ Gosper's summation algorithm.

GosperSummationMethod(E, V, R, P, Q): Exports == Impl where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: IntegralDomain
  P: PolynomialCategory(R, E, V)
  Q: Join(RetractableTo Fraction Integer, Field with
           (coerce: P -> %; numer : % -> P; denom : % -> P))

I ==> Integer
RN ==> Fraction I
PQ ==> SparseMultivariatePolynomial(RN, V)
RQ ==> Fraction PQ
Exports == with
  GospersMethod: (Q, V, () -> V) -> Union(Q, "failed")
  \[ \text{GospersMethod}(b, n, new) \text{ returns a rational function} \]
  \[ \text{such that} \ a(n) * rf(n) \text{ is the indefinite} \]
  \[ \text{sum of} \ a(n) \]
  \[ \text{with respect to upward difference on} \ a(n), \ i.e. \]
  \[ \text{where} \ b(n) = a(n)/a(n-1) \text{ is a rational function.} \]
  \[ \text{Returns "failed" if no such rational function} \ a(n) \]
  \[ \text{exists.} \]
  \[ \text{Note that} \ new \text{ is a nullary function returning a new} \]
  \[ \text{V every time.} \]
  \[ \text{The condition on} \ a(n) \text{ is that} \ a(n)/a(n-1) \]
  \[ \text{is a rational function of} \ a(n). \]
  \[ \text{---+} \ \text{sum}(a(n), n) = rf(n) * a(n). \]

Impl == add
  import PolynomialCategoryQuotientFunctions(E, V, R, P, Q)
  import LinearSystemMatrixPackage(RQ,Vector RQ,Vector RQ,Matrix RQ)

InnerGospersMethod: (RQ, V, () -> V) -> Union(RQ, "failed")
GoserPQR: (PQ, PQ, V, () -> V) -> List PQ
GosperDegBd: (PQ, PQ, V, () -> V) -> I
GosperF: (I, PQ, PQ, PQ, V, () -> V) -> Union(RQ, "failed")
linearAndNNIntRoot: (PQ, V) -> Union(I, "failed")
deg0: (PQ, V) -> I
  \[ \text{degree with} \ deg 0 = -1. \]
pCoef: (PQ, PQ) -> PQ
  \[ \text{pCoef}(p, a*b**2) \]
RF2QIfCan: Q -> Union(RQ, "failed")
UP2QIfCan: P -> Union(PQ,"failed")
RFQ2R : RQ -> Q
PQ2R : PQ -> Q
rat?: R -> Boolean

def0(p, v) == (zero? p => -1; degree(p, v))
rat? x == retractIfCan(x::P::Q)@Union(RN, "failed") case RN
RFQ2R f == PQ2R(numer f) / PQ2R(denom f)

GospersMethod(aquo, n, newV) ==
  ((q := RF2QIfCan aquo) case "failed") or
  ((u := InnerGospersMethod(q::RQ, n, newV)) case "failed") =>
  "failed"
  RFQ2R(u::RQ)
RF2QIfCan f ==
  (n := UP2QIfCan numer f) case "failed" => "failed"
  (d := UP2QIfCan denom f) case "failed" => "failed"
n::PQ / d::PQ

UP2QIfCan p ==
every?(rat?, coefficients p) =>
  map(x +-> x::PQ, y +-> (retractIfCan(y::P::Q)@Union(RN, "failed"))::RN::PQ,p)_
  $PolynomialCategoryLifting(E, V, R, P, PQ)
  "failed"

InnerGospersMethod(aquo, n, newV) ==
  -- 1. Define coprime polys an,anm1 such that
  -- an/anm1=a(n)/a(n-1)
  an := numer aquo
  anm1 := denom aquo

  -- 2. Define p,q,r such that
  -- a(n)/a(n-1) = (p(n)/p(n-1)) * (q(n)/r(n))
  -- and
  -- gcd(q(n), r(n+j)) = 1, for all j: NNI.
  pqr:= GosperPQR(an, anm1, n, newV)
  pn := first pqr; qn := second pqr; rn := third pqr

  -- 3. If the sum is a rational fn, there is a poly f with
  -- sum(a(n), n) = q(n+1)/p(n) * a(n) * f(n).

  -- 4. Bound the degree of f(n).
  (k := GosperDegBd(pn, qn, rn, n, newV)) < 0 => "failed"

  -- 5. Find a polynomial f of degree at most k, satisfying
  -- p(n) = q(n+1)*f(n) - r(n)*f(n-1)
  (ufn := GosperF(k, pn, qn, rn, n, newV)) case "failed" =>
    "failed"
  fn := ufn::RQ

  -- 6. The sum is q(n-1)/p(n)*f(n) * a(n). We leave out a(n).
  --qnm1 := eval(qn,n,n::PQ - 1)
  --qnm1/pn * fn
  qn1 := eval(qn,n,n::PQ + 1)
  qn1/pn * fn

GosperF(k, pn, qn, rn, n, newV) ==
  mv := newV(); mp := mv::PQ; np := n::PQ
  fn: PQ := +/\[mp**(i+1) * np**i for i in 0..k] fnminus1: PQ := eval(fn, n, np-1)
  qnplus1 := eval(qn, n, np+1)
  zro := qnplus1 * fn - rn * fnminus1 - pn
  zron := univariate(zro, n)
  dz := degree zron
  mat: Matrix RQ := zero(dz+1, (k+1)::NonNegativeInteger)
  vec: Vector RQ := new(dz+1, 0)
while zron ^= 0 repeat
    cz := leadingCoefficient zron
    dz := degree zron
    zron := reductum zron
    mz := univariate(cz, mv)
    while mz ^= 0 repeat
        cmz := leadingCoefficient(mz)::RQ
        dmz := degree mz
        mz := reductum mz
        dmz = 0 => vec(dz + minIndex vec) := -cmz
        qsetelt_!(mat, dz + minRowIndex mat, dmz + minColIndex(mat) - 1, cmz)
(soln := particularSolution(mat, vec)) case "failed" => "failed"
vec := soln::Vector RQ
(+/[np**i * vec(i + minIndex vec) for i in 0..k])@RQ

GosperPQR(an, anm1, n, newV) ==
np := n::PQ -- polynomial version of n
-- Initial guess.
qn := an
rn := anm1
-- Find all j: NNI giving common factors to q(n) and r(n+j).
j := newV()
qnj := eval(rn, n, np + j::PQ)
res := resultant(qn, qnj, n)
fres := factor(res)$MRationalFactorize(IndexedExponents V, V, I, PQ)
js := [rt::I for fe in factors fres | (rt := linearAndNNIntRoot(fe.factor, j)) case I]
-- For each such j, change variables to remove the gcd.
for rt in js repeat
    rtp:= rt::PQ -- polynomial version of rt
gn := gcd(qn, eval(rn,n,np+rtp))
qn := (qn exquo gn)::PQ
rn := (rn exquo eval(gn, n, np-rtp))::PQ
np := pn * */[eval(gn, n, np-i::PQ) for i in 0..rt-1]

GosperDegBd(pn, qn, rn, n, newV) ==
np := n::PQ
qnplus1 := eval(qn, n, np+1)
lplus := deg0(qnplus1 + rn, n)
lminus := deg0(qnplus1 - rn, n)
degp := deg0(pn, n)
k := degp - max(lplus-1, lminus)
lplus <= lminus => k
-- Find L(k), such that
-- \( p(n) = L(k)c[k]n^{k+1} + \ldots \)
-- To do this, write \( f(n) \) and \( f(n-1) \) symbolically.
-- \( f(n) = c[k]n^k + c[k-1]n^{k-1} + O(n^{k-2}) \)
-- \( f(n-1) = c[k]n^k + (c[k-1] - k*c[k])n^{k-1} + O(n^{k-2}) \)

\[
\begin{align*}
k_k & := \text{newV()}::\text{PQ} \\
ck & := \text{newV()}::\text{PQ} \\
ckm1 & := \text{newV()}::\text{PQ} \\
nk & := np*nkm1 \\
\text{headfn} & := ck*nk + ckm1*nkm1 \\
\text{headfnm1} & := ck*nk + (ckm1 - kk*ck)*nkm1 \\
\end{align*}
\]

Then \( p(n) = q(n+1)f(n) - r(n)f(n-1) \) gives \( L(k) \).

\[
\begin{align*}
pk & := qnplus1 * \text{headfn} - rn * \text{headfnm1} \\
lcpk & := \text{pCoef}(pk, ck*np*nkm1) \\
\end{align*}
\]

-- The degree \( bd \) is now given by \( k \), and the root of \( L \).

\[
\begin{align*}
k0 & := \text{linearAndNNIntRoot}(\text{lcpk}, \text{mainVariable}(kk)::V) \\
k0 & \text{ case "failed"} \Rightarrow k \\
\max(k0::I, k) & \\
\end{align*}
\]

\[
\begin{align*}
\text{pCoef}(p, \text{nom}) & = \\
\text{not monomial?} \text{ nom} & \Rightarrow \\
\text{error "pCoef requires a monomial 2nd arg"} \\
\text{vlist} & := \text{variables nom} \\
\text{for v in vlist while p} & \text{ ^= 0 repeat} \\
\text{unom} & := \text{univariate(nom, v)} \\
\text{pow} & := \text{degree unom} \\
\text{nom} & := \text{leadingCoefficient unom} \\
\text{up} & := \text{univariate}(p, v) \\
\text{p} & := \text{coefficient}(up, pow) \\
p & \\
\text{linearAndNNIntRoot}(mp, v) & = \\
p & := \text{univariate}(mp, v) \\
\text{degree p} & \text{ ^= 1} \Rightarrow "\text{failed}" \\
(p1 & := \text{retractIfCan}(\text{coefficient}(p, 1))::\text{Union}(\text{RN},"\text{failed}")) \\
\text{case "failed" or} \\
(p0 & := \text{retractIfCan}(\text{coefficient}(p, 0))::\text{Union}(\text{RN},"\text{failed}")) \\
\text{case "failed"} & \Rightarrow "\text{failed}" \\
\text{rt} & := -(p0::\text{RN})/(p1::\text{RN}) \\
\text{rt} & < 0 \text{ or } \text{denom rt} \text{ ^= 1} \Rightarrow "\text{failed}" \\
\text{numer rt} & \\
\end{align*}
\]

— GOSPER.dotabb —
"GOSPER" -> "PFECAT"

package GRDEF GraphicsDefaults

-- GraphicsDefaults.input --

\set break resume
\sys rm -f GraphicsDefaults.output
\spool GraphicsDefaults.output
\set message test on
\set message auto off
\clear all

--S 1 of 1
\show GraphicsDefaults
--E 1

\spool
\lisp (bye)

-- GraphicsDefaults.help --

====================================================================
GraphicsDefaults examples
====================================================================

TwoDimensionalPlotSettings sets global flags and constants
for 2-dimensional plotting.

See Also:
\o \show GraphicsDefaults

---
GraphicsDefaults (GRDEF)

Exports:
adaptive clipPointsDefault drawToScale maxPoints minPoints screenResolution

--- package GRDEF GraphicsDefaults ---

)abbrev package GRDEF GraphicsDefaults  
++ Author: Clifton J. Williamson
++ Date Created: 8 January 1990
++ Date Last Updated: 8 January 1990
++ Description:
++ TwoDimensionalPlotSettings sets global flags and constants
++ for 2-dimensional plotting.

GraphicsDefaults(): Exports == Implementation where
  B ==> Boolean
  I ==> Integer
  SF ==> DoubleFloat
  maxWidth ==> 1000
  maxHeight == 1000

Exports ==> with
  clipPointsDefault: () -> B
  ++ clipPointsDefault() determines whether or not automatic clipping is
  ++ to be done.
  drawToScale: () -> B
  ++ drawToScale() determines whether or not plots are to be drawn to scale.

clipPointsDefault: B -> B
  ++ clipPointsDefault(true) turns on automatic clipping;
  ++ \spad{clipPointsDefault(false)} turns off automatic clipping.
  ++ The default setting is true.

drawToScale: B -> B
  ++ drawToScale(true) causes plots to be drawn to scale.
  ++ \spad{drawToScale(false)} causes plots to be drawn so that they
  ++ fill up the viewport window.
++ The default setting is false.

--% settings from the two-dimensional plot package

adaptive: () -> B
  ++ adaptive() determines whether plotting will be done adaptively.
maxPoints: () -> I
  ++ maxPoints() returns the maximum number of points in a plot.
minPoints: () -> I
  ++ minPoints() returns the minimum number of points in a plot.
screenResolution: () -> I
  ++ screenResolution() returns the screen resolution n.

adaptive: B -> B
  ++ adaptive(true) turns adaptive plotting on;
  ++ \spad{adaptive(false)} turns adaptive plotting off.
maxPoints: I -> I
  ++ maxPoints() sets the maximum number of points in a plot.
minPoints: I -> I
  ++ minPoints() sets the minimum number of points in a plot.
screenResolution: I -> I
  ++ screenResolution(n) sets the screen resolution to n.

Implementation ==> add

--% global flags and constants

CLIPPOINTSDEFAULT : B := true
TOSCALE : B := false

--% functions

clipPointsDefault() == CLIPPOINTSDEFAULT
drawToScale() == TOSCALE

clipPointsDefault b == CLIPPOINTSDEFAULT := b
drawToScale b == TOSCALE := b

--% settings from the two-dimensional plot package

adaptive() == adaptive?()$Plot
minPoints() == minPoints()$Plot
maxPoints() == maxPoints()$Plot
screenResolution() == screenResolution()$Plot

adaptive b == setAdaptive(b)$Plot
minPoints n == setMinPoints(n)$Plot
maxPoints n == setMaxPoints(n)$Plot
screenResolution n == setScreenResolution(n)$Plot
package GRAY GrayCode

GrayCode examples

GrayCode provides a function for efficiently running through all subsets of a finite set, only changing one element by another one.

See Also:
  o )show GrayCode
GrayCode (GRAY)

Exports:
  firstSubsetGray  nextSubsetGray

— package GRAY GrayCode —

)abbrev package GRAY GrayCode
++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ References:
++ Henryk Minc: Evaluation of Permanents,
++ Nijenhuis and Wilf : Combinatorical Algorithms, Academic
++ S.G.Williamson, Combinatorics for Computer Science,
++ Description:
++ GrayCode provides a function for efficiently running
++ through all subsets of a finite set, only changing one element
++ by another one.

GrayCode: public == private where

  PI ==> PositiveInteger
  I  ==> Integer
  V  ==> Vector

public ==> with

  nextSubsetGray: (V V I,PI) -> V V I
  ++ nextSubsetGray(ww,n) returns a vector vv whose components
  ++ have the following meanings:
  ++ vv.1: a vector of length n whose entries are 0 or 1. This
  ++ can be interpreted as a code for a subset of the set 1,...,n;
  ++ vv.1 differs from ww.1 by exactly one entry;
++ vv.2.1 is the number of the entry of vv.1 which
++ will be changed next time; \br
++ vv.2.1 = n+1 means that vv.1 is the last subset;
++ trying to compute nextSubsetGray(vv) if vv.2.1 = n+1
++ will produce an error! \br
++
++ The other components of vv.2 are needed to compute
++ nextSubsetGray efficiently.
++ Note that this is an implementation of [Williamson, Topic II, 3.54,
++ p. 112] for the special case r1 = r2 = \ldots = rn = 2;
++ Note that nextSubsetGray produces a side-effect, i.e.
++ nextSubsetGray(vv) and vv := nextSubsetGray(vv)
++ will have the same effect.

firstSubsetGray: PI -> V V I
++ firstSubsetGray(n) creates the first vector ww to start a
++ loop using nextSubsetGray(ww,n)

private ==> add

firstSubsetGray(n : PI) ==
vv : V V I := new(2,[])
vv.1 := new(n,0) @ V I
vv.2 := new(n+1,1) @ V I
for i in 1..(n+1) repeat
  vv.2.i := i
vv

nextSubsetGray(vv : V V I,n : PI) ==
subs : V I := vv.1 -- subset
lab : V I := vv.2 -- labels
c : I := lab(1) -- element which is to be changed next
lab(1):= 1
if subs.c = 0 then subs.c := 1
else subs.c := 0
lab.c := lab(c+1)
lab(c+1) := c+1
vv

GRAY.dotabb

"GRAY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GRAY"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"GRAY" -> "IVECTOR"
package GBF GroebnerFactorizationPackage

— GroebnerFactorizationPackage.input —

)set break resume
)sys rm -f GroebnerFactorizationPackage.output
)spool GroebnerFactorizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 3
mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := [
  [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y],
  [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0]
]

--R
|--
|--
|--
|--
|--
|--

(1) | 8 8 |
|--
|--
|--
|--
|--
|--

Type: SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],Fraction(Integer)))

--E 1

--S 2 of 3

eq := determinant mfzn

--R

(2)

22 25 2 22 2 388 250 25 2 250 14575

Type: DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))

--E 2
groebnerFactorize [eq, eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])]

\[
\begin{align*}
g & \rightarrow \begin{bmatrix} 22 & 22 & 22 & 121 \\
2 & 22 & 25 & 2 & 22 & 25 & 22 & 2 & 388 & 250 \\
3 & 9 & 3 & 9 & 9 & 9 & 27 \\
3 & 9 & 3 & 9 & 27 & 9 & 27 & 81 \\
\end{bmatrix} \\
\rightarrow & \begin{bmatrix} 21994 & 2 & 21994 & 4427 & 463 \\
21994 & 2 & 25 & 463 \\
5625 & 5625 & 675 & 87 \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
2 & 2 & 6 & 18 & 3 & 9 \\
25 & 11 & 11 & 11 & 11 & 5 & 5 & 5 \\
9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
19 & 5 & 5 \\
3 & 3 & 3 \\
\end{bmatrix} \\
\rightarrow & \begin{bmatrix} x + y - -----, y - -----, + ---, z - ---- \\
5625 & 5625 & 675 & 87 \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
2 & 2 & 6 & 18 & 3 & 9 \\
25 & 11 & 11 & 11 & 11 & 5 & 5 & 5 \\
9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
19 & 5 & 5 \\
3 & 3 & 3 \\
\end{bmatrix} \\
\rightarrow & \begin{bmatrix} x + y - -----, y - -----, + ---, z - ---- \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
2 & 2 & 6 & 18 & 3 & 9 \\
25 & 11 & 11 & 11 & 11 & 5 & 5 & 5 \\
9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
19 & 5 & 5 \\
3 & 3 & 3 \\
\end{bmatrix} \\
\rightarrow & \begin{bmatrix} x + y - -----, y - -----, + ---, z - ---- \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
2 & 2 & 6 & 18 & 3 & 9 \\
25 & 11 & 11 & 11 & 11 & 5 & 5 & 5 \\
9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
19 & 5 & 5 \\
3 & 3 & 3 \\
\end{bmatrix} \\
\rightarrow & \begin{bmatrix} x + y - -----, y - -----, + ---, z - ---- \\
5625 & 5625 & 675 & 87 \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
2 & 2 & 6 & 18 & 3 & 9 \\
25 & 11 & 11 & 11 & 11 & 5 & 5 & 5 \\
9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
19 & 5 & 5 \\
3 & 3 & 3 \\
\end{bmatrix} \end{align*}
\]

Type: List(List(DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))))

--- E 3 
)spool 
)lisp (bye)

--- GroebnerFactorizationPackage.help ---

GroebnerFactorizationPackage provides the function "groebnerFactor" which uses the factorization routines of Axiom to factor each polynomial under consideration while doing the groebner basis algorithm. Then it writes the ideal as an intersection of ideals determined by the irreducible factors. Note that the whole ring may occur as well as other redundancies. We also use the fact, that from the second factor on we can assume that the preceding factors are not equal to 0 and we divide all polynomials.
under considerations by the elements of this list of "nonZeroRestrictions".

The result is a list of groebner bases, whose union of solutions of the corresponding systems of equations is the solution of the system of equation corresponding to the input list. The term ordering is determined by the polynomial type used.

Suggested types include
- DistributedMultivariatePolynomial
- HomogeneousDistributedMultivariatePolynomial
- GeneralDistributedMultivariatePolynomial

Solving systems of polynomial equations with the Groebner basis algorithm can often be very time consuming because, in general, the algorithm has exponential run-time. These systems, which often come from concrete applications, frequently have symmetries which are not taken advantage of by the algorithm. However, it often happens in this case that the polynomials which occur during the Groebner calculations are reducible. Since Axiom has an excellent polynomial factorization algorithm, it is very natural to combine the Groebner and factorization algorithms.

GroebnerFactorizationPackage exports the groebnerFactorize operation which implements a modified Groebner basis algorithm. In this algorithm, each polynomial that is to be put into the partial list of the basis is first factored. The remaining calculation is split into as many parts as there are irreducible factors. Call these factors p1,...,pN. In the branches corresponding to p2,...,pN, the factor p1 can be divided out, and so on. This package also contains operations that allow you to specify the polynomials that are not zero on the common roots of the final Groebner basis.

Here is an example from chemistry. In a theoretical model of the cyclohexan C6H12, the six carbon atoms each sit in the center of gravity of a tetrahedron that has two hydrogen atoms and two carbon atoms at its corners. We first normalize and set the length of each edge to 1. Hence, the distances of one fixed carbon atom to each of its immediate neighbours is 1. We will denote the distances to the other three carbon atoms by x, y and z.

A. Dress developed a theory to decide whether a set of points and distances between them can be realized in an n-dimensional space. Here, of course, we have n = 3.

```
mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := _
  [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y],_
    [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
  +0 1 1 1 1 1 +
  |     |     |
  | 8 8 |
```
Type: SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],
Fraction Integer))

For the cyclohexan, the distances have to satisfy this equation.

\[
\begin{align*}
\text{eq} &:= \text{determinant mfn}\n\begin{array}{cccccccc}
2 & 2 & 2 & 2 & 2 & 2 & 2 & 388 & 250 & 25 & 2 & 250 & 14575 \\
- x y + & x y - & x + & x y - & x y - & x y - & x - & y - & y + ----
\end{array} \\
3 & 9 & 3 & 9 & 27 & 9 & 27 & 81 \\
\text{Type: DistributedMultivariatePolynomial([x,y,z],Fraction Integer)}
\end{align*}
\]

They also must satisfy the equations given by cyclic shifts of the indeterminates.

\[
\text{groebnerFactorize } \{\text{eq,eval(eq, [x,y,z],[y,z,x])}, \
\text{eval(eq,[x,y,z],[z,x,y])}\} \\
\{ 22 & 22 & 22 & 121 \\
x z - & x z + & x + y z - & y + y z + & y z + & y z + & y z + & y + z + & z + -- \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 9 & 9 & 9 & 9 & 27 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 388 & 250 & 25 & 2 & 250 & 14575 \\
y z - & y z + & y z + & y z + & y z + & y z + & y z + & y + z + & z + -- \\
3 & 3 & 3 & 9 & 9 & 9 & 9 & 9 & 27 & 27 & 27 & 27 & 81 \\
\}, \\
21994 & 2 & 21994 & 4427 & 463 \\
x + y - & y - & y + & z - & - - \\
5625 & 5625 & 675 & 87 \\
2 & 1 & 11 & 5 & 265 & 2 & 38 & 265 \\
x + & x z - & x + & z + & y - & z, z - & z + - - \\
2 & 2 & 6 & 18 & 3 & 9
\]
The union of the solutions of this list is the solution of our original problem. If we impose positivity conditions, we get two relevant ideals. One ideal is zero-dimensional, namely \( x = y = z = \frac{11}{3} \), and this determines the "boat" form of the cyclohexan. The other ideal is one-dimensional, which means that we have a solution space given by one parameter. This gives the "chair" form of the cyclohexan. The parameter describes the angle of the "back of the chair."

\textbf{groebnerFactorize} has an optional Boolean-valued second argument. When it is true partial results are displayed, since it may happen that the calculation does not terminate in a reasonable time. See the source code for \texttt{GroebnerFactorizationPackage} in \texttt{groebf.spad.pamphlet} for more details about the algorithms used.

See Also:
\begin{itemize}
\item \texttt{o )display operations groebnerFactorize}
\item \texttt{o )show GroebnerFactorizationPackage}
\item \texttt{o )show GroebnerPackage}
\item \texttt{o )show EuclideanGroebnerBasisPackage}
\end{itemize}
package GBF GroebnerFactorizationPackage

+ Author: H. Michael Moeller, Johannes Grabmeier
+ Date Created: 24 August 1989
+ Date Last Updated: 01 January 1992
+ Description:
  + \spadtype{GroebnerFactorizationPackage} provides the function
  + \spadtype{groebnerFactor} which uses the factorization routines of Axiom to
  + factor each polynomial under consideration while doing the groebner basis
  + algorithm. Then it writes the ideal as an intersection of ideals
  + determined by the irreducible factors. Note that the whole ring may
  + occur as well as other redundancies. We also use the fact, that from the
  + second factor on we can assume that the preceding factors are
  + not equal to 0 and we divide all polynomials under considerations
  + by the elements of this list of "nonZeroRestrictions".
  + The result is a list of groebner bases, whose union of solutions
  + of the corresponding systems of equations is the solution of
  + the system of equation corresponding to the input list.
  + The term ordering is determined by the polynomial type used.
  + Suggested types include
  + \spadtype{DistributedMultivariatePolynomial},
  + \spadtype{HomogeneousDistributedMultivariatePolynomial},
  + \spadtype{GeneralDistributedMultivariatePolynomial}.

GroebnerFactorizationPackage(Dom, Expon, VarSet, Dpol): T == C where

Dom : Join(EuclideanDomain,CharacteristicZero)
Expon : OrderedAbelianMonoidSup
VarSet : OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)
MF ==> MultivariateFactorize(VarSet,Expon,Dom,Dpol)
sugarPol ==> Record(totdeg: NonNegativeInteger, pol : Dpol)
critPair ==> Record(lcmfij: Expon,totdeg: NonNegativeInteger, poli: Dpol, polj: Dpol )
L ==> List
B ==> Boolean
NNI ==> NonNegativeInteger
OUT ==> OutputForm

T ==> with

factorGroebnerBasis : L Dpol -> L L Dpol
  ++ factorGroebnerBasis(basis) checks whether the basis contains
  ++ reducible polynomials and uses these to split the basis.
factorGroebnerBasis : (L Dpol, Boolean) -> L L Dpol
  ++ factorGroebnerBasis(basis,info) checks whether the basis contains
  ++ reducible polynomials and uses these to split the basis.
  ++ If argument info is true, information is printed about
  ++ partial results.
groebnerFactorize : (L Dpol, L Dpol) -> L L Dpol
++ groebnerFactorize(listOfPolys, nonZeroRestrictions) returns
++ a list of groebner basis. The union of their solutions
++ is the solution of the system of equations given by listOfPolys
++ under the restriction that the polynomials of nonZeroRestrictions
++ don't vanish.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.

groebnerFactorize : (L Dpol, L Dpol, Boolean) -> L L Dpol
++ groebnerFactorize(listOfPolys, nonZeroRestrictions, info) returns
++ a list of groebner basis. The union of their solutions
++ is the solution of the system of equations given by listOfPolys
++ under the restriction that the polynomials of nonZeroRestrictions
++ don't vanish.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor in place of p.
++ If argument info is true, information is printed about
++ partial results.

groebnerFactorize : L Dpol -> L L Dpol
++ groebnerFactorize(listOfPolys) returns
++ a list of groebner bases. The union of their solutions
++ is the solution of the system of equations given by listOfPolys.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.
++
++X mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := _
++X  [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y],
++X  [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
++X eq := determinant mfzn
++X groebnerFactorize _
++X [eq,eval(eq, [x,y,z],[y,z,x]), eval(eq,x,y,z],[z,x,y])]

groebnerFactorize : (L Dpol, Boolean) -> L L Dpol
++ groebnerFactorize(listOfPolys, info) returns
++ a list of groebner bases. The union of their solutions
++ is the solution of the system of equations given by listOfPolys.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.
++ If info is true, information is printed about partial results.

C ==> add

import GroebnerInternalPackage(Dom,Expon,VarSet,Dpol)
-- next to help compiler to choose correct signatures:
info: Boolean
-- signatures of local functions

newPairs : (L sugarPol, Dpol) -> L critPair
++ newPairs(lp, p) constructs list of critical pairs from the list of
++ lp of input polynomials and a given further one p.
++ It uses criteria M and T to reduce the list.
updateCritPairs : (L critPair, L critPair, Dpol) -> L critPair
++ updateCritPairs(lcP1,lcP2,p) applies criterion B to lcP1 using
++ p. Then this list is merged with lcP2.
updateBasis : (L sugarPol, Dpol, NNI) -> L sugarPol
++ updateBasis(li,p,deg) every polynomial in li is dropped if
++ its leading term is a multiple of the leading term of p.
++ The result is this list enlarged by p.
createGroebnerBases : (L sugarPol, L Dpol, L Dpol, L Dpol, L critPair,_,
++ L L Dpol, Boolean) -> L L Dpol
++ createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,
++ lcP,listOfBases): This function is used to be called from
++ groebnerFactorize.
++ basis: part of a Groebner basis, computed so far
++ redPols: Polynomials from the ideal to be used for reducing,
++ we don't throw away polynomials
++ nonZeroRestrictions: polynomials not zero in the common zeros
++ of the polynomials in the final (Groebner) basis
++ inputPolys: assumed to be in descending order
++ lcP: list of critical pairs built from polynomials of the
++ actual basis
++ listOfBases: Collects the (Groebner) bases constructed by this
++ recursive algorithm at different stages.
++ we print info messages if info is true
createAllFactors: Dpol -> L Dpol
++ factor reduced critpair polynomial

-- implementation of local functions

createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,_,
++ lcP, listOfBases, info) ==
doSplitting? : B := false
terminateWithBasis : B := false
allReducedFactors : L Dpol := []
nP : Dpol -- actual polynomial under consideration
p : Dpol -- next polynomial from input list
h : Dpol -- next polynomial from critical pairs
stopDividing : Boolean
-- STEP 1 do the next polynomials until a splitting is possible
-- In the first step we take the first polynomial of "inputPolys"
-- if empty, from list of critical pairs "lcP" and do the following:
-- Divide it, if possible, by the polynomials from "nonZeroRestrictions".
-- We factorize it and reduce each irreducible factor with respect to
-- "basis". If 0$Dpol occurs in the list we update the list and continue
-- with next polynomial.
-- If there are at least two (irreducible) factors
-- in the list of factors we finish STEP 1 and set a boolean variable
-- to continue with STEP 2, the splitting step.
-- If there is just one of it, we do the following:
-- If it is 1$Dpol we stop the whole calculation and put
-- [1$Dpol] into the listOfBases
- Otherwise we update the "basis" and the other lists and continue
-- with next polynomial.

while (not doSplitting?) and (not terminateWithBasis) repeat
terminateWithBasis := (null inputPolys and null lcP)
not terminateWithBasis => -- still polynomials left
-- determine next polynomial "nP"
nP :=
not null inputPolys =>
p := first inputPolys
inputPolys := rest inputPolys
-- we know that p is not equal to 0 or 1, but, although,
-- the inputPolys and the basis are ordered, we cannot assume
-- that p is reduced w.r.t. basis, as the ordering is only quasi
-- and we could have equal leading terms, and due to factorization
-- polynomials of smaller leading terms, hence reduce p first:
hMonic redPol(p,redPols)
-- now we have inputPolys empty and hence lcP is not empty:
-- create S-Polynomial from first critical pair:
h := sPol first lcP
lcP := rest lcP
hMonic redPol(h,redPols)

nP = 1$Dpol =>
basis := [[0,1$Dpol]$sugarPol]
terminateWithBasis := true

-- if "nP" ^= 0, then we continue, otherwise we determine next "nP"
nP ^= 0$Dpol =>
-- now we divide "nP", if possible, by the polynomials
-- from "nonZeroRestrictions"
for q in nonZeroRestrictions repeat
stopDividing := false
until stopDividing repeat
nPq := nP exquo q
stopDividing := (nPq case "failed")
if not stopDividing then nP := autoCoerce nPq
stopDividing := stopDividing or zero? degree nP

zero? degree nP =>
basis := [[0,1$Dpol]$sugarPol]
terminateWithBasis := true -- doSplitting? is still false

-- a careful analysis has to be done, when and whether the
-- following reduction and case nP=1 is necessary

nP := hMonic redPol(nP,redPols)
zero? degree nP =>
basis := [[0,1$Dpol]$sugarPol]
terminateWithBasis := true -- doSplitting? is still false

-- if "nP" ^= 0, then we continue, otherwise we determine next "nP"
nP ^= 0$Dpol =>

-- now we factorize "nP", which is not constant
irreducibleFactors : L Dpol := createAllFactors(nP)

-- if there are more than 1 factors we reduce them and split
(doSplitting? := not null rest irreducibleFactors) =>

-- and reduce and normalize the factors
for fnP in irreducibleFactors repeat
fnP := hMonic redPol(fnP,redPols)
-- no factor reduces to 0, as then "fP" would have been
-- reduced to zero,
-- but 1 may occur, which we will drop in a later version.
allReducedFactors := cons(fnP, allReducedFactors)
-- end of "for fnP in irreducibleFactors repeat"

-- we want that the smaller factors are dealt with first
allReducedFactors := reverse allReducedFactors
-- now the case of exactly 1 factor, but certainly not
-- further reducible with respect to "redPols"
nP := first irreducibleFactors
-- put "nP" into "basis" and update "lcP" and "redPols":
lcP : L critPair := updateCritPairs(lcP,newPairs(basis,nP),nP)
basis := updateBasis(basis,nP,virtualDegree nP)
redPols := concat(redPols,nP)
-- end of "while not doSplitting? and not terminateWithBasis repeat"

-- STEP 2 splitting step

doSplitting? =>
for fnP in allReducedFactors repeat
if fnP ^= 1$Dpol
then
newInputPolys : L Dpol := _
sort((x,y) +-> degree x > degree y ,cons(fnP,inputPolys))
listOfBases := createGroebnerBases(basis, redPols, _
nonZeroRestrictions,newInputPolys,lcP,listOfBases,info)
-- update "nonZeroRestrictions"
nonZeroRestrictions := cons(fnP,nonZeroRestrictions)
else
if info then
messagePrint("we terminated with [1]")$OUT
listOfBases := cons([1$Dpol],listOfBases)
-- we finished with all the branches on one level and hence
-- finished this call of createGroebnerBasis. Therefore
-- we terminate with the actual "listOfBasis" as
-- everything is done in the recursions
listOfBases
-- end of "doSplitting? =>"

-- STEP 3 termination step

-- we found a groebner basis and put it into the list "listOfBases"
-- (auto)reduce each basis element modulo the others
newBasis :=
minGbasis(sort((x,y)+->degree x > degree y,[p.pol for p in basis]))
-- now check whether the normalized basis again has reducible
-- polynomials, in this case continue splitting!
if info then
messagePrint("we found a groebner basis and check whether it ")$OUT
messagePrint("contains reducible polynomials")$OUT
print(newBasis::OUT)$OUT
-- here we should create an output form which is reusable by the system
-- print(convert(newBasis::OUT)$InputForm :: OUT)$OUT
removeDuplicates append(factorGroebnerBasis(newBasis, info), listOfBases)

createAllFactors(p: Dpol) ==
loF : L Dpol := [el.fctr for el in factorList factor(p)$MF]
sort((x,y) +-> degree x < degree y, loF)
newPairs(lp : L sugarPol,p : Dpol) ==
totdegreeOfp : NNI := virtualDegree p
-- next list lcP contains all critPair constructed from
-- p and and the polynomials q in lp
lcP: L critPair := _
--[[sup(degree q, degreeOfp), q, p]$critPair for q in lp]
[makeCrit(q, p, totdegreeOfp) for q in lp]
-- application of the criteria to reduce the list lcP
critMTonD1 sort(critpOrder,lcP)
updateCritPairs(oldListOfcritPairs, newListOfcritPairs, p)==
updatD (newListOfcritPairs, critBonD(p,oldListOfcritPairs))
updateBasis(lp, p, deg) == updatF(p,deg,lp)

-- exported functions
factorGroebnerBasis basis == factorGroebnerBasis(basis, false)

factorGroebnerBasis (basis, info) ==
  foundAReducible : Boolean := false
  for p in basis while not foundAReducible repeat
    -- we use fact that polynomials have content 1
    foundAReducible := 1 < #[@el.fctr for el in factorList factor(p)$MF]
  not foundAReducible =>
    if info then messagePrint("factorGroebnerBasis: no reducible polynomials in this basis")$OUT [basis]
    -- improve! Use the fact that the irreducible ones already
    -- build part of the basis, use the done factorizations, etc.
    if info then messagePrint("factorGroebnerBasis: we found reducible polynomials and continue splitting")$OUT
    createGroebnerBases([],[],[],basis,[],[],info)

groebnerFactorize(basis, nonZeroRestrictions) ==
  groebnerFactorize(basis, nonZeroRestrictions, false)

groebnerFactorize(basis, nonZeroRestrictions, info) ==
  basis = [] => [basis]
  basis := remove((x:Dpol):Boolean +-> (x = 0$Dpol), basis)
  basis = [] => [[0$Dpol]]
  -- normalize all input polynomial
  basis := [hMonic p for p in basis]
  member?(1$Dpol,basis) => [[1$Dpol]]
  basis := sort((x,y) +-> degree x > degree y, basis)
  createGroebnerBases([],[],[],basis,[],[],info)

groebnerFactorize(basis) == groebnerFactorize(basis, [], false)
groebnerFactorize(basis,info) == groebnerFactorize(basis, [], info)

package GBINTERN GroebnerInternalPackage
--- GroebnerInternalPackage.input ---

)set break resume
)sys rm -f GroebnerInternalPackage.output
)spool GroebnerInternalPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GroebnerInternalPackage
--E 1

)spool
)lisp (bye)

-----

--- GroebnerInternalPackage.help ---

====================================================================
GroebnerInternalPackage examples
====================================================================

This package provides low level tools for Groebner basis computations

See Also:
 o )show GroebnerInternalPackage

-----

GroebnerInternalPackage (GBINTERN)
Exports:
credPol  critB  critBonD  critM  critMTonD1
critMonD1  critpOrder  critT  fprindINFO  gbasis
hMonic  lepol  makeCrit  minGbasis  prnb
prindINFO  prinpolINFO  prinshINFO  redPo  redPol
sPol  updatD  updatF  virtualDegree

| package GBINTERN GroebnerInternalPackage |

)abbrev package GBINTERN GroebnerInternalPackage
++ Description:
++ This package provides low level tools for Groebner basis computations

GroebnerInternalPackage(Dom, Expon, VarSet, Dpol): T == C where
Dom: GcdDomain
Expon: OrderedAbelianMonoidSup
VarSet: OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)

NNI ==> NonNegativeInteger

------ Definition of Record critPair and Prinp

critPair ==> Record( lcmfij: Expon, totdeg: NonNegativeInteger, poli: Dpol, polj: Dpol )
sugarPol ==> Record( totdeg: NonNegativeInteger, pol : Dpol)
Prinp  ==> Record( ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol, tc:Integer,rc:Dpol,trc:Integer,tF:Integer,tD:Integer)

T== with

credPol: (Dpol, List(Dpol)) -> Dpol
++ credPol \undocumented
redPol: (Dpol, List(Dpol)) -> Dpol
++ redPol \undocumented
gbasis: (List(Dpol), Integer, Integer) -> List(Dpol)
++ gbasis \undocumented
critT: critPair -> Boolean
++ critT \undocumented
critM: (Expon, Expon) -> Boolean
++ critM \undocumented
critB: (Expon, Expon, Expon) -> Boolean
++ critB \undocumented
critBonD: (Dpol, List(critPair)) -> List(critPair)
++ critBonD \undocumented
critMTonD1: (List(critPair)) -> List(critPair)
++ critMTonD1 \undocumented
critMonD1: (Expon, List(critPair)) -> List(critPair)
++ critMonD1 \undocumented
redPo: (Dpol, List(Dpol) ) -> Record(poly:Dpol, mult:Dom)
++ redPo \undocumented
hMonic: Dpol -> Dpol
++ hMonic \undocumented
updatF: (Dpol, NNI, List(sugarPol) ) -> List(sugarPol)
++ updatF \undocumented
sPol: critPair -> Dpol
++ sPol \undocumented
updatD: (List(critPair), List(critPair)) -> List(critPair)
++ updatD \undocumented
minGbasis: List(Dpol) -> List(Dpol)
++ minGbasis \undocumented
lepol: Dpol -> Integer
++ lepol \undocumented
prinshINFO : Dpol -> Void
++ prinshINFO \undocumented
prindINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
++ prindINFO \undocumented
fprindINFO: (critPair, Dpol, Dpol, Integer,Integer,Integer
, Integer) -> Integer
++ fprindINFO \undocumented
prinpolINFO: List(Dpol) -> Void
++ prinpolINFO \undocumented
prinb: Integer-> Void
++ prinb \undocumented
critpOrder: (critPair, critPair) -> Boolean
++ critpOrder \undocumented
makeCrit: (sugarPol, Dpol, NonNegativeInteger) -> critPair
++ makeCrit \undocumented
virtualDegree : Dpol -> NonNegativeInteger
++ virtualDegree \undocumented

C== add
Ex ==> OutputForm
import OutputForm

-------  Definition of intermediate functions
if Dpol has totalDegree: Dpol -> NonNegativeInteger then
  virtualDegree p == totalDegree p
else
  virtualDegree p == 0

------- ordering of critpairs

critpOrder(cp1,cp2) ==
cp1.totdeg < cp2.totdeg => true
cp2.totdeg < cp1.totdeg => false
cp1.lcmfij < cp2.lcmfij

------- creating a critical pair
makeCrit(sp1, p2, totdeg2) ==
  p1 := sp1.pol
  deg := sup(degree(p1), degree(p2))
  e1 := subtractIfCan(deg, degree(p1))::Expon
  e2 := subtractIfCan(deg, degree(p2))::Expon
  tdeg := max(sp1.totdeg + virtualDegree(monomial(1,e1)),
              totdeg2 + virtualDegree(monomial(1,e2)))
  [deg, tdeg, p1, p2]$critPair

------ calculate basis

gbasis(Pol: List(Dpol), xx1: Integer, xx2: Integer ) ==
  D, D1: List(critPair)
  --------- create D and Pol

  Pol1:= sort((z1,z2) +-> degree z1 > degree z2, Pol)
  basPols:= updatF(hMonic(first Pol1),virtualDegree(first Pol1),[])
  Pol1:= rest(Pol1)
  D:= nil
  while _^ null Pol1 repeat
    h:= hMonic(first(Pol1))
    Pol1:= rest(Pol1)
    toth := virtualDegree h
    D1:= [makeCrit(x,h,toth) for x in basPols]
    D:= updatD(critMTonD1(sort(critpOrder, D1)),
                 critBonD(h,D))
    basPols:= updatF(h,toth,basPols)
  D:= sort(critpOrder, D)
  xx:= xx2
  -------- loop

  redPols := [x.pol for x in basPols]
  while _^ null D repeat
    D0:= first D
    s:= hMonic(sPol(D0))
    D:= rest(D)
    h:= hMonic(redPol(s,redPols))
    if xx1 = 1 then
      prinshINFO(h)
    h = 0 =>
    if xx2 = 1 then
      prinINFO(D0,s,h,# basPols, # D,xx)
    xx:= 2
    " go to top of while "
    degree(h) = 0 =>
    D:= nil
    if xx2 = 1 then
      prinINFO(D0,s,h,# basPols, # D,xx)
    xx:= 2
    basPols:= updatF(h,0,[])
leave "out of while"
D1:= [makeCrit(x,h,D0.totdeg) for x in basPols]
D:= updatD(critMTonD1(sort(critpOrder, D1)),
critBonD(h,D))
basPols:= updatF(h,D0.totdeg,basPols)
redPols := concat(redPols,h)
if xx2 = 1 then
  prinDI({D0,s,h,# basPols, # D,xx})
xx:= 2
Pol := [x.pol for x in basPols]
if xx2 = 1 then
  prinpolINFO(Pol)
messagePrint(" THE GROEBNER BASIS POLYNOMIALS")
if xx1 = 1 and xx2 ^= 1 then
  messagePrint(" THE GROEBNER BASIS POLYNOMIALS")
Pol

--------------------------------------
--- erase multiple of e in D2 using crit M
critMonD1(e: Expon, D2: List(critPair))==
  null D2 => nil
  x:= first(D2)
  critM(e, x.lcmfij) => critMonD1(e, rest(D2))
  cons(x, critMonD1(e, rest(D2)))

-----------------------------
--- reduce D1 using crit T and crit M
critMTonD1(D1: List(critPair))==
  null D1 => nil
  f1:= first(D1)
  s1:= #(D1)
  cT1:= critT(f1)
  s1= 1 and cT1 => nil
  s1= 1 => D1
  e1:= f1.lcmfij
  r1:= rest(D1)
  e1 = (first r1).lcmfij =>
    cT1 =>
      critMTonD1(cons(f1, rest(r1)))
      critMTonD1(r1)
    D1 := critMonD1(e1, r1)
    cT1 =>
      critMTonD1(D1)
      cons(f1, critMTonD1(D1))

-----------------------------
--- erase elements in D fullfilling crit B
critBonD(h: Dpol, D: List(critPair)) ==
  null D => nil
  x := first(D)
  critB(degree(h), x.lcmfij, degree(x.poli), degree(x.polj)) =>
    critBonD(h, rest(D))
  cons(x, critBonD(h, rest(D)))

-------------

--- concat F and h and erase multiples of h in F

updatF(h: Dpol, deg: NNI, F: List(sugarPol)) ==
  null F => [[deg, h]]
  f1 := first(F)
  critM(degree(h), degree(f1.pol)) => updatF(h, deg, rest(F))
  cons(f1, updatF(h, deg, rest(F)))

-------------

--- concat ordered critical pair lists D1 and D2

updatD(D1: List(critPair), D2: List(critPair)) ==
  null D1 => D2
  null D2 => D1
  d11 := first(D1)
  d12 := first(D2)
  critpOrder(d11, d12) => cons(d11, updatD(D1.rest, D2))
  cons(d12, updatD(D1, D2.rest))

-------------

--- remove gcd from pair of coefficients

gcdCo(c1: Dom, c2: Dom): Record(co1: Dom, co2: Dom) ==
  d := gcd(c1, c2)
  [(c1 exquo d):: Dom, (c2 exquo d):: Dom]

--- calculate S-polynomial of a critical pair

sPol(p: critPair) ==
  Tij := p.lcmfij
  fi := p.poli
  fj := p.polj
  cc := gcdCo(leadingCoefficient fi, leadingCoefficient fj)
  reductum(fi)*monomial(cc.co2, subtractIfCan(Tij, degree fi):: Expon) -
  reductum(fj)*monomial(cc.co1, subtractIfCan(Tij, degree fj):: Expon)
--- reduce critpair polynomial mod F
--- iterative version

redPo(s: Dpol, F: List(Dpol)) ==
m: Dom := 1
Fh := F
while _^ ( s = 0 or null F ) repeat
  f1 := first(F)
s1 := degree(s)
e: Union(Expon, "failed")
  (e:= subtractIfCan(s1, degree(f1))) case Expon ⇒
    cc:=gcdCo(leadingCoefficient f1, leadingCoefficient s)
    s:=cc.co1*reductum(s) - monomial(cc.co2,e)*reductum(f1)
    m := m*cc.co1
    F:= Fh
    F:= rest F
  [s,m]

redPol(s: Dpol, F: List(Dpol)) == credPol(redPo(s,F).poly,F)

-----------------------------
--- crit T true, if e1 and e2 are disjoint

critT(p: critPair) == p.lcmfij = (degree(p.poli) + degree(p.polj))

-----------------------------
--- crit M - true, if lcm#2 multiple of lcm#1

critM(e1: Expon, e2: Expon) ==
en: Union(Expon, "failed")
  (en:=subtractIfCan(e2, e1)) case Expon

-----------------------------
--- crit B - true, if eik is a multiple of eh and eik = equal
--- lcm(eh,ei) and eik = equal lcm(eh,ek)
critB(eh:Expon, eik:Expon, ei:Expon, ek:Expon) ==
  critM(eh, eik) and (eik = sup(eh, ei)) and (eik = sup(eh, ek))

-----------------------------
--- make polynomial monic case Domain a Field

hMonic(p: Dpol) ==
p = 0 => p
-- inv(leadingCoefficient(p))*p
primitivePart p
--- reduce all terms of $h$ mod $F$ (iterative version)

credPol(h: Dpol, F: List(Dpol) ) ==
null F => h
h0:Dpol:= monomial(leadingCoefficient h, degree h)
while (h:=reductum h) ^= 0 repeat
    hred:= redPo(h, F)
    h := hred.poly
    h0:=(hred.mult)*h0 + monomial(leadingCoefficient(h),degree h)
    h0

---- calculate minimal basis for ordered $F$

minGbasis(F: List(Dpol)) ==
null F => nil
newbas := minGbasis rest F
cons(hMonic credPol( first(F), newbas),newbas)

---- calculate number of terms of polynomial

lepol(p1:Dpol)==
n: Integer
n:= 0
while p1 ^= 0 repeat
    n:= n + 1
    p1:= reductum(p1)
n
---- print blanc lines

prinb(n: Integer)==
for x in 1..n repeat
    messagePrint(" ")

---- print reduced critpair polynom

prinshINFO(h: Dpol)==
prinb(2)
messagePrint(" reduced Critpair - Polynom :")
prinb(2)
print(h::Ex)
prinb(2)
---- print info string

prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer, i2:Integer, n:Integer) ==

ll: List Prinp
a: Dom
cpi:= cp.poli
cpj:= cp.polj
if n = 1 then
  prinb(1)
  messagePrint("you choose option -info- ")
  messagePrint("abbrev. for the following information strings are")
  messagePrint(" ci => Leading monomial for critpair calculation")
  messagePrint(" tci => Number of terms of polynomial i")
  messagePrint(" cj => Leading monomial for critpair calculation")
  messagePrint(" tcj => Number of terms of polynomial j")
  messagePrint(" c => Leading monomial of critpair polynomial")
  messagePrint(" tc => Number of terms of critpair polynomial")
  messagePrint(" rc => Leading monomial of redcritpair polynomial")
  messagePrint(" trc => Number of terms of redcritpair polynomial")
  messagePrint(" tF => Number of polynomials in reduction list F")
  messagePrint(" tD => Number of critpairs still to do")
  prinb(4)
n:= 2
prinb(1)
a:= 1
ph = 0 =>
  ps = 0 =>
    ll:= [[monomial(a,degree(cpi)),lepol(cpi),
        monomial(a,degree(cpj)),
        lepol(cpj),ps,0,ph,0,i1,i2]$Prinp]
    print(ll::Ex)
    prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
        monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
        lepol(ps), ph,0,i1,i2]$Prinp]
    print(ll::Ex)
    prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
        monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
        lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2]$Prinp]
    print(ll::Ex)
    prinb(1)
n
---- print the groebner basis polynomials

prinpolINFO(pl: List(Dpol)) ==
  n: Integer
  n := # pl
  prinb(1)
  n = 1 =>
    messagePrint(" There is 1 Groebner Basis Polynomial ")
    prinb(2)
    messagePrint(" There are ")
    prinb(1)
    print(n::Ex)
    prinb(1)
    messagePrint(" Groebner Basis Polynomials. ")
    prinb(2)

fprindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1: Integer, i2: Integer, i3: Integer, n: Integer) ==
  ll: List Prinpp
  a: Dom
  cpi:= cp.poli
  cpj:= cp.polj
  if n = 1 then
    prinb(1)
    messagePrint("you choose option -info- ")
    messagePrint("abbrev. for the following information strings are")
    messagePrint(" ci => Leading monomial for critpair calculation")
    messagePrint(" tci => Number of terms of polynomial i")
    messagePrint(" cj => Leading monomial for critpair calculation")
    messagePrint(" tcj => Number of terms of polynomial j")
    messagePrint(" c => Leading monomial of critpair polynomial")
    messagePrint(" tc => Number of terms of critpair polynomial")
    messagePrint(" rc => Leading monomial of redcritpair polynomial")
    messagePrint(" trc => Number of terms of redcritpair polynomial")
    messagePrint(" tF => Number of polynomials in reduction list F")
    messagePrint(" tD => Number of critpairs still to do")
    messagePrint(" tDF => Number of subproblems still to do")
  prinb(4)
  n:= 2
  prinb(1)
  a:= 1
  ph = 0 =>
    ps = 0 =>
      ll:= [[monomial(a,degree(cpi)),lepol(cpi),
        monomial(a,degree(cpj)),
        lepol(cpj),ps,0,ph,0,i1,i2,i3]$Prinpp]
      print(ll::Ex)
      prinb(1)
      n
\[ll := [\text{monomial}(a, \text{degree}(cpi)), \text{lepol}(cpi), \text{monomial}(a, \text{degree}(cpj)), \text{lepol}(cpj), \text{monomial}(a, \text{degree}(ps)), \text{lepol}(ps), \text{ph}, 0, i1, i2, i3] \text{Prinpp}\]
\[\text{print}(ll::\text{Ex})\]
\[\text{prinb}(1)\]
\[n\]
\[ll := [\text{monomial}(a, \text{degree}(cpi)), \text{lepol}(cpi), \text{monomial}(a, \text{degree}(cpj)), \text{lepol}(cpj), \text{monomial}(a, \text{degree}(ps)), \text{lepol}(ps), \text{monomial}(a, \text{degree}(ph)), \text{lepol}(ph), i1, i2, i3] \text{Prinpp}\]
\[\text{print}(ll::\text{Ex})\]
\[\text{prinb}(1)\]
\[n\]

\text{—— GBINTERN.dotabb ——}

"GBINTERN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GBINTERN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GBINTERN" -> "PFECAT"

\text{—— package GB GroebnerPackage ——}

\text{—— GroebnerPackage.input ——}

\text{)}set break resume
\text{)}sys rm -f GroebnerPackage.output
\text{)}spool GroebnerPackage.output
\text{)}set message test on
\text{)}set message auto off
\text{)}clear all
\text{--S 1 of 24}
\text{s1:DMP([w,p,z,t,s,b],FRAC(INT)) := 45*p + 35*s - 165*b - 36}
\text{--R}
\text{--R (1) 45p + 35s - 165b - 36}
\text{--R Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))}
\text{--E 1}

\text{--S 2 of 24}
\text{s2:DMP([w,p,z,t,s,b],FRAC(INT)) := 35*p + 40*z + 25*t - 27*s}
\text{--R}
\text{--R (2) 35p + 40z + 25t - 27s}
s3 := \[15w + 25ps + 30z - 18t - 165b^2\]

s4 := \[-9w + 15pt + 20zs\]

s5 := \[wp + 2zt - 11b^3\]

s6 := \[99w - 11bs + 3b^2\]

s7 := \[b^2 + \frac{33}{50}b + \frac{2673}{10000}\]

sn7 := \[s1, s2, s3, s4, s5, s6, s7\]
reduced Critpair - Polynom :

```
3 1026 2 5424 2 2529 1326807 12717 660717
145 3625 725 362500 6250 3625000
```

reduced Critpair - Polynom :

```
2 91248294 2 6550614 7087292937 20020838931
128176525 5127061 12817652500 12817652500
```

reduced Critpair - Polynom :

```
2 4746183626079988 1015195815329760 30723564870033201
987357073521193 987357073521193 24683926838029825
```

reduced Critpair - Polynom :

```
0
```

reduced Critpair - Polynom :
THE GROEBNER BASIS POLYNOMIALS

\begin{verbatim}
Reduced Critpair - Polynom:
\end{verbatim}
groebner(sn7,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R
--R [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 3]]
--R
--R [[ci= w,tci= 3,cj= w,tcj= 5,c= p t,tc= 6,rc= t s,trc= 8,tF= 5,tD= 2]]
--R
--R [[ci= w,tci= 3,cj= w,tcj= 3,c= p t,tc= 4,rc= t, trc= 6,tF= 5,tD= 2]]
--R
--R 3
--R [[ci= t s,tci= 8,cj= t,tcj= 6,c= t b,tc= 9,rc= s ,trc= 7,tF= 6,tD= 1]]
--R
--R 2
--R [[ci= w p,tci= 3,cj= w,tcj= 3,c= p s b,tc= 4,rc= s b,trc= 6,tF= 7,tD= 2]]
--R
--R 2 2 2 2
--R [[ci= b ,tci= 3,cj= s b,tcj= 6,c= s b,tc= 6,rc= s ,trc= 5,tF= 6,tD= 2]]
--R
--R 2 2 2
--R [[ci= s b,tci= 6,cj= s ,tcj= 5,c= s ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R 3 2 2
--R [[ci= s ,tci= 7,cj= s ,tcj= 5,c= s b,tc= 6,rc= s b,trc= 4,tF= 7,tD= 2]]
--R
--R 2
--R [[ci= b ,tci= 3,cj= s b,tcj= 4,c= s b,tc= 4,rc= s ,trc= 3,tF= 6,tD= 2]]
There are 6 Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

(11)

<table>
<thead>
<tr>
<th>19</th>
<th>1323</th>
<th>31</th>
<th>153</th>
<th>49</th>
<th>1143</th>
<th>37</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>b</td>
<td>p</td>
<td>b</td>
<td>z</td>
<td>b</td>
<td>t</td>
<td>b</td>
</tr>
<tr>
<td>120</td>
<td>20000</td>
<td>18</td>
<td>200</td>
<td>36</td>
<td>2000</td>
<td>15</td>
<td>250</td>
</tr>
<tr>
<td>s</td>
<td>b</td>
<td></td>
<td>b</td>
<td></td>
<td>b</td>
<td>t</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
<td>33</td>
<td>2673</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))

you choose option -info-

abbrev. for the following information strings are

ci => Leading monomial for critpair calculation

tci => Number of terms of polynomial i

cj => Leading monomial for critpair calculation

tcj => Number of terms of polynomial j

c => Leading monomial of critpair polynomial

tc => Number of terms of critpair polynomial

rc => Leading monomial of redcritpair polynomial
--R  trc => Number of terms of redcritpair polynomial
--R  tF => Number of polynomials in reduction list F
--R  tD => Number of critpairs still to do
--R
--R
--R
--R
--R [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 66  603  278  2  11  672  2277  415881
--R 29  1450  435  29  725  7250  725000
--R
--R
--R
--R [[ci= w,tci= 3,cj= w,tcj= 5,c= p t,tc= 6,rc= t s,trc= 8,tF= 5,tD= 2]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 100  2  160  104  37  79
--R 189  63  63  105  125
--R
--R
--R
--R [[ci= w,tci= 3,cj= w,tcj= 3,c= p t,tc= 4,rc= t,ttrc= 6,tF= 5,tD= 2]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3  1026  2  5424  2  2529  1326807  12717  660717
--R 145  3625  725  362500  6250  3625000
--R
--R
--R
--R [[ci= t s,tci= 8,cj= t,tcj= 6,c= t b,tc= 9,rc= s,trc= 7,tF= 6,tD= 1]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R 2 91248294 2 6550614 7087292937 20020838931
--R s b +--------- s - ------- s b + --------- s - ----------- b
--R 128176525 5127061 12817652500 12817652500
--R +
--R 37595502243
--R - ---------
--R 51270610000
--R
--R
--R
--R
--R
--R 2
--R [[ci= w p, tci= 3, cj= w, tcj= 3, c= p s b, tc= 4, rc= s b, trc= 6, tF= 7, tD= 2]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R 2 4746183626079988 1015195815329760 30723564870033201
--R s - --------------- s b - --------------- s - --------------- b
--R 987357073521193 987357073521193 24683926838029825
--R +
--R 3696123458901625353
--R - ---------------
--R 2468392683802982500
--R
--R
--R
--R 2
--R [[ci= b, tci= 3, cj= s b, tcj= 6, c= s b, tc= 6, rc= s, trc= 5, tF= 6, tD= 2]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R 2
--R [[ci= s b, tci= 6, cj= s, tcj= 5, c= s, tc= 7, rc= 0, trc= 0, tF= 6, tD= 1]]
--R
--R
--R
CHAPTER 8. CHAPTER G

---R reduced Critpair - Polynom :
---R
---R
---R 1682737360807663182513471 1262793163581645698534964
---R s b + -------------------------- s - ------------------------- b
---R 23063714246644859914108300 5765928561661214978527075
---R +
---R 91594345205981119652436033
---R --------------------------------------
---R 144148214041530374463176875
---R ---
---R
---R
---R [3  2  2]
---R [[ci= s ,tci= 7,cj= s ,tcj= 5,c= s b,tc= 6,rc= s b,trc= 4,tF= 7,tD= 2]]
---R
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R 5  9
---R s - - b - ---
---R 2  200
---R
---R
---R
---R [2]
---R [[ci= b ,tci= 3,cj= s b,tcj= 4,c= s b,tc= 4,rc= s,trc= 3,tF= 6,tD= 2]]
---R
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R 0
---R
---R
---R
---R [0]
---R [[ci= s b,tci= 4,cj= s,tcj= 3,c= s,tc= 4,rc= 0,trc= 0,tF= 6,tD= 1]]
---R
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R 0
---R
---R
---R
There are 6 Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

(12)
\[
\begin{align*}
w + \frac{19}{120} b + \frac{1323}{20000}, \\
p - \frac{12}{20000} b + \frac{31}{18} b + \frac{153}{2000}, \\
z + \frac{49}{36} b + \frac{1143}{2000}, \\
t - \frac{37}{15} b + \frac{27}{250}.
\end{align*}
\]

Type: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))

hs1:HDMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36

(13) 45p + 35s - 165b - 36

Type: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))

hs2:HDMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s

(14) 35p + 40z + 25t - 27s

Type: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))

hs3:HDMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2

(15) 25p s - 165b + 15w + 30z - 18t

Type: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))

hs4:HDMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s

(16) 15p t + 20z s - 9w

Type: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
hs5: \( \text{HDMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3} \)

hs6: \( \text{HDMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2} \)

hs7: \( \text{HDMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000} \)

hsn7:=[hs1,hs2,hs3,hs4,hs5,hs6,hs7]

groebner(hsn7)
groebner(hsn7, "redcrit")

---

reduced Critpair - Polynom:

5 61 77 7
z + - t - --- s + -- b + --
8 45 24 10

reduced Critpair - Polynom:

2 216 189 78 99 10557
s - --- w + ---- t - -- s + --- b - ----- 5 100 25 500 12500

reduced Critpair - Polynom:

66 17541 5886 10588 9273 8272413
t s - -- t b - ------ w + ----- t - ------ s - ------ b - ------- 29 725 3625 3625 36250 7250000

reduced Critpair - Polynom:

2 28 44 143 962712 420652 5166944
t + -- w s - -- w b + --- t b - ------ w + ----- t - ------ s 45 15 725 18125 90625 815625

+ 5036339 83580953
---

- 

---

---

---
reduced Critpair - Polynom:

```
33  297  81
w b + -- w + ----- s - ----- b
50  10000  10000
```

reduced Critpair - Polynom:

```
21  33  6723  2031  104247
w s + --- t b - --- w + ----- s - ----- b + -------
100  250  50000  25000  500000
```

reduced Critpair - Polynom:

```
2373  41563  17253  578853  258751  11330361
w t + ---- t b - ----- w + ------ t + ------- s - ------- b + -------
7250  36250  290000  7250000  36250000  362500000
```

reduced Critpair - Polynom:

```
0
```

reduced Critpair - Polynom:

```
51061712  91248294  1516761889  481096937  5789482077
```

reduced Critpair - Polynom:

```
5127061  128176525  1922647875  1281765250  51270610000
```

reduced Critpair - Polynom:

```
```
Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:

Reduced Critpair - Polynom:
reduced Critpair - Polynom:

0

reduced Critpair - Polynom:

0

reduced Critpair - Polynom:

0

reduced Critpair - Polynom:

0

reduced Critpair - Polynom:

0

reduced Critpair - Polynom:

0
--R
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R THE GROEBNER BASIS POLYNOMIALS
--R
--R (22)
--R 2 33 2673 19 1323 31 153 49 1143
--R [b +-- b + ------, w +--- b + ------, p --- b - ---, z +-- b + ----,
--R 50 10000 120 20000 18 200 36 2000
--R 37 27 5 9
--R t --- b + ---, a - - b - ---]
--R 15 250 2 200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 22

--S 23 of 24
groebner(hsn7,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R
--R
--R
--R 

[[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 5]]
--R
--R
--R 2
--R 

[[ci= p,s,tci= 5,cj= p,tcj= 4,c= z,s,tc= 7,rc= s,trc= 6,tF= 5,tD= 5]]
--R
--R
--R 

[[ci= p,t,tci= 3,cj= p,tcj= 4,c= z,t,tc= 5,rc= t,s,trc= 7,tF= 6,tD= 6]]
There are 6 Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

(23)

2  33  2673  19  1323  31  153  49  1143
[\text{b} + \text{--} \text{b} + \text{-----}, \text{w} + \text{---} \text{b} + \text{-----}, \text{p} - \text{--} \text{b} - \text{---}, \text{z} + \text{--} \text{b} + \text{-----},
\text{50} 10000 120 20000 18 200 36 2000
\text{t} - \text{--} \text{b} - \text{---}, \text{s} - \text{--} \text{b} - \text{---}]

\text{E 23}

groebner(hsn7,"redcrit","info")

reduced Critpair - Polynom:
you choose option -info-
abbrev. for the following information strings are
- ci => Leading monomial for critpair calculation
- tci => Number of terms of polynomial i
- cj => Leading monomial for critpair calculation
- tcj => Number of terms of polynomial j
- c => Leading monomial of critpair polynomial
- tc => Number of terms of critpair polynomial
- rc => Leading monomial of redcritpair polynomial
- trc => Number of terms of redcritpair polynomial
- tF => Number of polynomials in reduction list F
- tD => Number of critpairs still to do

[[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 5]]

reduced Critpair - Polynom :

2  216  189  78   99  10557
s  -  ---  w +  ---  t  -  ---  s  +  ---  b  -  ----- 
5   100  25  500  12500

[[ci= p s,tci= 5,cj= p,tcj= 4,c= z s,tc= 7,rc= s,trc= 6,tF= 5,tD= 5]]

reduced Critpair - Polynom :

66  17541  5886  10588  9273  8272413
ts  -  ---  t  b  -  -----  w  +  -----  t  -  -----  s  -  -----  b  -  ------
29   725  3625  3625  36250  7250000
--R [[ci= p t, tci= 3, cj= p, tcj= 4, c= z t, t= 5, rc= t s, trc= 7, tF= 6, tD= 6]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R 2 28 44 143 962712 420652 5166944
--R t + -- w s - -- w b + --- t b - ------- w + ------- t - ------- s
--R 45 15 725 18125 90625 815625
--R +
--R 5036339 83580953
--R ------- b - -------
--R 5437500 90625000
--R
--R
--R
--R
--R
--R
--R 3 2 2
--R [[ci= b , tci= 3, cj= b , tcj= 3, c= w p, tc= 4, rc= t , trc= 9, tF= 7, tD= 6]]
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 33 297 81
--R w b + -- w + ----- s - ----- b
--R 50 10000 10000
--R
--R
--R
--R
--R
--R
--R
--R 2 3
--R [[ci= s b, tci= 3, cj= b , tcj= 3, c= b , tc= 4, rc= w b, trc= 4, tF= 8, tD= 7]]
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R 2 2
--R [[ci= s b, tci= 3, cj= s , tcj= 6, c= s b , tc= 7, rc= w s, trc= 6, tF= 9, tD= 9]]
--R
--R
```
reduced Critpair - Polynom:

2
[[ci= s b, tci= 3, cj= t s, tcj= 7, c= t b, t= 7, r= w t, tr= 7, tF= 10, tD= 11]]

reduced Critpair - Polynom:

2
[[ci= p s, tci= 5, cj= s b, tcj= 3, c= p b, t= 6, r= 0, tr= 0, tF= 10, tD= 10]]

reduced Critpair - Polynom:

2
[[ci= s , tci= 6, cj= t s, tcj= 7, c= t s b, t= 10, r= t b, tr= 6, tF= 11, tD= 13]]

reduced Critpair - Polynom:

w + t - s + b
99240140318722273
```
PACKAGE GB GROEBNER

997

--R 1226727352610000
--R
--R
--R
--R 2
--R [[[ci= b ,tci= 3,cj= t b,tcj= 6,c= w b,tc= 6,rc= w,tcrc= 5,tF= 9,tD= 14]]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R 0
--R
--R
--R
--R
--R 2
--R [[[ci= b ,tci= 3,cj= w b,tcj= 4,c= s b,tc= 3,rc= 0,tcrc= 0,tF= 9,tD= 13]]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R 2
--R [[[ci= s b,tci= 3,cj= t b,tcj= 6,c= t b ,tc= 7,rc= t,tcrc= 4,tF= 7,tD= 11]]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R 5 9
--R s - - b - ---
--R 2 200
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R 2
--R [[[ci= s b,tci= 3,cj= w b,tcj= 4,c= w b ,tc= 5,rc= s,tcrc= 3,tF= 6,tD= 9]]]
--R
--R
--R
--R reduced Critpair - Polynom :
reduced Critpair - Polynom :

\[
\begin{align*}
&\text{reduced Critpair - Polynom :} \\
&\text{red Critpair - Polynom :} \\
&\text{red Critpair - Polynom :} \\
&\text{red Critpair - Polynom :} \\
&\text{red Critpair - Polynom :}
\end{align*}
\]
--R
--R
--R 2
--R [ci= s, tci= 6, cj= s, tcj= 3, c= s b, tc= 6, rc= 0, trc= 0, tF= 6, tD= 4]
--R
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R
--R 2
--R [ci= t s, tci= 7, cj= t, tcj= 4, c= s t, tc= 8, rc= 0, trc= 0, tF= 6, tD= 3]
--R
--R
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R
--R
--R [ci= w s, tci= 6, cj= w, tcj= 5, c= t s, tc= 8, rc= 0, trc= 0, tF= 6, tD= 2]
--R
--R
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R
--R 2
--R [ci= t, tci= 9, cj= t, tcj= 4, c= w s, tc= 9, rc= 0, trc= 0, tF= 6, tD= 1]
--R
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R
--R 2
There are 6 Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

There are 6 Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

Groebner Package computes groebner bases for polynomial ideals. The basic computation provides a distinguished set of generators for polynomial ideals over fields. This basis allows an easy test for membership: the operation normalForm returns zero on ideal members. When the provided coefficient domain, Dom, is not a field, the result is equivalent to considering the extended ideal with Fraction(Dom) as coefficients, but considerably more efficient since all calculations are performed in Dom.

Additional arguments "info" and "redcrit" can be given to provide incremental information during computation.

Argument "info" produces a computational summary for each s-polynomial. Argument "redcrit" prints out the reduced critical pairs.

The term ordering is determined by the polynomial type used.

Suggested types include
Example to call groebner:

```
s1:DMP[w,p,z,t,s,b]RN:= 45*p + 35*s - 165*b - 36
s2:DMP[w,p,z,t,s,b]RN:= 35*p + 40*z + 25*t - 27*s
s3:DMP[w,p,z,t,s,b]RN:= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
s4:DMP[w,p,z,t,s,b]RN:= -9*w + 15*p*t + 20*z*s
s5:DMP[w,p,z,t,s,b]RN:= w*p + 2*z*t - 11*b**3
s6:DMP[w,p,z,t,s,b]RN:= 99*w - 11*b*s + 3*b**2
s7:DMP[w,p,z,t,s,b]RN:= b**2 + 33/50*b + 2673/10000

sn7:=[s1,s2,s3,s4,s5,s6,s7]

groebner(sn7,info)
```

groebner calculates a minimal Groebner Basis
all reductions are TOTAL reductions

To get the reduced critical pairs do:

```
groebner(sn7,"redcrit")
```

You can get other information by calling:

```
groebner(sn7,"info")
```

which returns:

```
ci => Leading monomial for critpair calculation
tci => Number of terms of polynomial i

cj => Leading monomial for critpair calculation
tcj => Number of terms of polynomial j

c => Leading monomial of critpair polynomial
tc => Number of terms of critpair polynomial

rc => Leading monomial of redcritpair polynomial
trc => Number of terms of redcritpair polynomial

tF => Number of polynomials in reduction list F
tD => Number of critpairs still to do
```

See Also:
- `)display operations groebner`
- `)show GroebnerPackage`
- `)show DistributedMultivariatePolynomial`
- `)show HomogeneousDistributedMultivariatePolynomial`
- `)show EuclideanGroebnerBasisPackage`
GroebnerPackage (GB)

Exports:
groebner  normalForm

— package GB GroebnerPackage —

)abbrev package GB GroebnerPackage
++ Authors: Gebauer, Trager
++ Date Created: 12-1-86
++ Date Last Updated: 2-28-91
++ Description:
++ \spadtype{GroebnerPackage} computes groebner
++ bases for polynomial ideals. The basic computation provides a distinguished
++ set of generators for polynomial ideals over fields. This basis allows an
++ easy test for membership: the operation \spadfun{normalForm}
++ returns zero on ideal members. When the provided coefficient domain, Dom,
++ is not a field, the result is equivalent to considering the extended
++ ideal with \spadtype{Fraction(Dom)} as coefficients, but considerably more
++ efficient since all calculations are performed in Dom. Additional
++ argument "info" and "redcrit" can be given to provide incremental
++ information during computation. Argument "info" produces a computational
++ summary for each s-polynomial.
++ Argument "redcrit" prints out the reduced critical pairs. The term ordering
++ is determined by the polynomial type used. Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

GroebnerPackage(Dom, Expon, VarSet, Dpol): T == C where

Dom:  GcdDomain
Expon: OrderedAbelianMonoidSup
VarSet: OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)

T== with

groebner: List(Dpol) -> List(Dpol)
++ groebner(lp) computes a groebner basis for a polynomial ideal
++ generated by the list of polynomials lp.
++
++ s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++ s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++ s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++ s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++ s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++ s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
++ s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
++ sn7:=[s1,s2,s3,s4,s5,s6,s7]
++ groebner(sn7)

++ groebner: (List(Dpol), String ) -> List(Dpol)
++ groebner(lp, infoflag) computes a groebner basis
++ for a polynomial ideal
++ generated by the list of polynomials lp.
++ Argument infoflag is used to get information on the computation.
++ If infoflag is "info", then summary information
++ is displayed for each s-polynomial generated.
++ If infoflag is "redcrit", the reduced critical pairs are displayed.
++ If infoflag is any other string,
++ no information is printed during computation.
++
++ s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++ s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++ s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++ s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++ s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++ s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
++ s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
++ sn7:=[s1,s2,s3,s4,s5,s6,s7]
++ groebner(sn7,"info")
++ groebner(sn7,"redcrit")

++ groebner: (List(Dpol), String, String ) -> List(Dpol)
++ groebner(lp, "info", "redcrit") computes a groebner basis
++ for a polynomial ideal generated by the list of polynomials lp,
++ displaying both a summary of the critical pairs considered ("info")
++ and the result of reducing each critical pair ("redcrit").
++ If the second or third arguments have any other string value,
++ the indicated information is suppressed.
++
++ s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++ s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++ X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++ X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++ X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++ X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
++ X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
++ X sn7:=[s1,s2,s3,s4,s5,s6,s7]
++ X groebner(sn7,"info","redcrit")

if Dom has Field then
normalForm: (Dpol, List(Dpol)) -> Dpol
++ normalForm(poly,gb) reduces the polynomial poly modulo the
++ precomputed groebner basis gb giving a canonical representative
++ of the residue class.
C== add
import OutputForm
import GroebnerInternalPackage(Dom,Expon,VarSet,Dpol)

if Dom has Field then
monicize(p: Dpol):Dpol ==
-- one?(lc := leadingCoefficient p) => p
((lc := leadingCoefficient p) = 1) => p
inv(lc)*p

normalForm(p : Dpol, l : List(Dpol)) : Dpol ==
redPol(p,map(monicize,l))

------ MAIN ALGORITHM GROEBNER ------------------------
groebner( Pol: List(Dpol) ) ==
Pol=[] => Pol
Pol:=[x for x in Pol | x ^= 0]
Pol=[] => [0]
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,0,0)))
groebner( Pol: List(Dpol), xx1: String) ==
Pol=[] => Pol
Pol:=[x for x in Pol | x ^= 0]
Pol=[] => [0]
xx1 = "redcrit" =>
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,0)))
xx1 = "info" =>
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,2,1)))
messagePrint(" ")
messagePrint("WARNING: options are - redcrit and/or info - ")
messagePrint(" you didn’t type them correct")
messagePrint(" please try again")
messagePrint(" ")
[]
groebner( Pol: List(Dpol), xx1: String, xx2: String) ==
Pol=[] => Pol
Pol:=[x for x in Pol | x ^= 0]
Pol=[] => [0]
(xx1 = "redcrit" and xx2 = "info") or
(xx1 = "info" and xx2 = "redcrit") =>
  minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,1)))
xx1 = "redcrit" and xx2 = "redcrit" =>
  minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,0)))
xx1 = "info" and xx2 = "info" =>
  minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,2,1)))
messagePrint(" ")
messagePrint("WARNING: options are - redcrit and/or info - ")
messagePrint(" you didn't type them correctly")
messagePrint(" please try again ")
messagePrint(" ")
[]
--- GroebnerSolve.help ---

GroebnerSolve examples

Solve systems of polynomial equations using Groebner bases. Total order Groebner bases are computed and then converted to lex ones. This package is mostly intended for internal use.

See Also:
  o )show GroebnerSolve

---

GroebnerSolve (GROEBSOL)

Exports:
genéricPosition groebSolve testDim

--- package GROEBSOL GroebnerSolve ---

)abbrev package GROEBSOL GroebnerSolve
++ Author : P.Gianni, Summer ’88, revised November ’89
++ Description:
++ Solve systems of polynomial equations using Groebner bases
++ Total order Groebner bases are computed and then converted to lex ones
This package is mostly intended for internal use.

GroebnerSolve(lv,F,R) : C == T

where
R : GcdDomain
F : GcdDomain
lv : List Symbol

NNI ==> NonNegativeInteger
I ==> Integer
S ==> Symbol

OV ==> OrderedVariableList(lv)
IES ==> IndexedExponents Symbol

DP ==> DirectProduct(#lv,NonNegativeInteger)
DPoly ==> DistributedMultivariatePolynomial(lv,F)
HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,F)
SUP ==> SparseUnivariatePolynomial(DPoly)
L ==> List
P ==> Polynomial

C == with
  groebSolve : (L DPoly,L OV) -> L L DPoly
  testDim : (L HDPoly,L OV) -> Union(L HDPoly,"failed")
  genericPosition : (L DPoly, L OV) -> Record(dpolys:L DPoly, coords: L I)

T == add
  import PolToPol(lv,F)
  import GroebnerPackage(F,DP,OV,DPoly)
  import GroebnerInternalPackage(F,DP,OV,DPoly)
  import GroebnerPackage(F,HDP,OV,HDPoly)
  import LinGroebnerPackage(lv,F)

  nv:NNI:=#lv
--- test if \( f \) is power of a linear mod (rad \( lpol \)) ---
--- \( f \) is monic ---

\[
testPower(uf:SUP,x:OV,lpol:L\ DPoly) : \text{Union}(DPoly,"failed") ==
\]
\[
df := \text{degree}(uf)
\]
\[
\text{trailp} := \text{coefficient}(uf,(df-1)\text{::NNI})
\]
\[
\text{testquo} := \text{trailp} \text{ exquo} (df::F) \text{ case } "\text{failed}" \Rightarrow "\text{failed}"
\]
\[
\text{trailp} := \text{testquo}::DPoly
\]
\[
\text{gg} := \text{gcd}(\text{lc} := \text{leadingCoefficient}(uf), \text{trailp})
\]
\[
\text{trailp} := (\text{trailp} \text{ exquo} \text{gg})::DPoly
\]
\[
\text{lc} := (\text{lc} \text{ exquo} \text{gg})::DPoly
\]
\[
\text{linp} := \text{monomial}(\text{lc},1\text{NNI})\text{SUP} + \text{monomial}(\text{trailp},0\text{NNI})\text{SUP}
\]
\[
g := \text{multivariate}(\text{uf} - \text{linp}^\text{df},x)
\]
\[
\text{redPol}(g,lpol) ^= 0 \Rightarrow "\text{failed}"
\]
\[
\text{multivariate}(\text{linp},x)
\]

-- is the 0-dimensional ideal \( I \) in general position ? --
--- internal function ---

\[
testGenPos(lpol:L\ DPoly,lvar:L\ OV):\text{Union}(L\ DPoly,"failed") ==
\]
\[
\text{rlpol} := \text{reverse} lpol
\]
\[
f := \text{rlpol}.\text{first}
\]
\[
\text{#lvar} = 1 \Rightarrow [f]
\]
\[
\text{rlvar} := \text{rest} \text{ reverse} \text{ lvar}
\]
\[
\text{newlpol} :\text{List}(DPoly) := [f]
\]
\[
\text{for} f \ \text{in} \ \text{rlpol}.\text{rest} \ \text{repeat}
\]
\[
x := \text{first} \ \text{rlvar}
\]
\[
\text{fi} := \text{univariate}(f,x)
\]
\[
\text{if} \ (\text{mainVariable leadingCoefficient} \text{fi} \ \text{case} "\text{failed}") \ \text{then}
\]
\[
\text{if} \ ((g := \text{testPower(fi,x,newlpol)}) \ \text{case} "\text{failed}"
\]
\[
\text{then} \ \text{return} "\text{failed}"
\]
\[
\text{newlpol} := \text{concat}(\text{redPol}(g::DPoly,newlpol),newlpol)
\]
\[
\text{rlvar} := \text{rest} \ \text{rlvar}
\]
\[
\text{else} \ \text{if} \ \text{redPol}(f,newlpol) ^= 0 \ \text{then} \ \text{return}"\text{failed}"
\]
\[
\text{newlpol}
\]

-- change coordinates and out the ideal in general position ----

\[
\text{genPos}(lp:L\ DPoly,lvar:L\ OV): \text{Record}(\text{polys:L HDpoly, lpolys:L DPoly,}
\]
\[
\text{coord:L I, univp:HDpoly}) ==
\]
\[
\text{rlvar} := \text{reverse} \ lvar
\]
\[
\text{lnp} := [\text{dmpToHdmp}(f) \ \text{for} \ f \ \text{in} \ \text{lp}]
\]
\[
\text{x} := \text{first} \ \text{rlvar}; \text{rlvar} := \text{rest} \ \text{rlvar}
\]
\[
\text{testfail} := \text{true}
\]
\[
\text{for} \ \text{count} \ \text{in} 1.. \ \text{while} \ \text{testfail} \ \text{repeat}
\]
\[
\text{ranvals} := [1 + (\text{random}()\text{::I} \ \text{rem} \ (\text{count}\# \ \text{lvar}))) \ \text{for} \ vv \ \text{in} \ \text{rlvar}]
\]
\[
\text{val} := \text{[/rv*(vv::HDpoly)}
\]
\[
\text{for} \ vv \ \text{in} \ \text{rlvar} \ \text{for} \ vv \ \text{in} \ \text{ranvals}]
\]
\[
\text{val} := \text{val+x::HDpoly}
\]
\[
\text{gb} := \text{groebner} \ \text{gb}
\]
gbt:=totolex gb
\{(gb1:=testGenPos(gbt,lvar)) case "failed"="try again"

testfail:=false
\[[gb,gbt,ranvals,dmpToHdmp(last (gb1::L DPoly))]\]

genericPosition(lp:L DPoly,lvar:L OV) ==
nans:=genPos(lp,lvar)
\[nans.lpolys, nans.coord\]

---- select the univariate factors
select(lup:L L HDPoly) : L L HDPoly ==
lup=[] => list []
\[:cons(f, lsel) for lsel in select lup.rest \] for f in lup.first\]

---- in the non generic case, we compute the prime ideals ----
---- associated to leq, basis is the algebra basis ----
findCompon(leq:L HDPoly,lvar:L OV):L L DPoly ==
teq:=totolex(leq)
#teq = #lvar => [teq]
-- ^((teq1:=testGenPos(teq,lvar)) case "failed") => [teq1::L DPoly]
gp:=genPos(teq,lvar)
lgp:= gp.univp
g:HDPoly:=gp.univp
fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDP,R,F,HDPoly)
lfact:=[ff.factor for ff in factors(fg::Factored(HDPoly))]"result": L L HDPoly := []
#lfact=1 => [teq]
for tfact in lfact repeat
\ tlfact:=concat(tfact,lgp)
\ result:=concat(tlfact,result)\]
ranvals:L I:=gp.coord
rlvar:=reverse lvar
x:=first rlvar
rlvar:=rest rlvar
val:=+\[rv*(vv::HDPoly) for vv in rlvar for rv in ranvals\]
val:=(x::HDPoly)-val
ans:=[totolex groebner \[elt(univariate(p,x),val) for p in lp\]
\ for lp in result]\]
\[ll for ll in ans | ll^=[1]\]

zeroDim?(lp: List HDPoly,lvar:L OV) : Boolean ==
empty? lp => false
n:NNI := #lvar
#lp < n => false
lvint1 := lvar
for f in lp while not empty?(lvint1) repeat
\ g:= f - reductum f
\ x:=mainVariable(g)::OV
\ if ground?(leadingCoefficient(univariate(g,x))) then
\ lvint1 := remove(x, lvint1)\]
empty? lvint1

-- general solve, gives an error if the system not 0-dimensional
groebSolve(leq: L DPoly,lvar:L OV) : L L DPoly ==
lnp:=[dmpToHdmp(f) for f in leq]
leq1:=groebner lnp
#(leq1) = 1 and first(leq1) = 1 => list empty()
 ~(zeroDim?(leq1,lvar)) =>
   error "system does not have a finite number of solutions"
-- add computation of dimension, for a more useful error
basis:=computeBasis(leq1)
lup:L HDPoly:=[]
llfact:L Factored(HDPoly):=[]
for x in lvar repeat
  g:=minPol(leq1,basis,x)
  fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDPoly,R,F,HDPoly)
  llfact:=concat(fg::Factored(HDPoly),llfact)
  if degree(g,x) = #basis then leave "stop factoring"
result: L L DPoly := []
-- selecting a factor from the lists of the univariate factors
lfact:=select [[ff.factor for ff in factors llf]
  for llf in llfact]
for tfact in lfact repeat
  tfact:=groebner concat(tfact,leq1)
  tfact=[1] => "next value"
  result:=concat(result,findCompon(tfact,lvar))
result

-- test if the system is zero dimensional
testDim(leq : L HDPoly,lvar : L OV) : Union(L HDPoly,"failed") ==
leq1:=groebner leq
#(leq1) = 1 and first(leq1) = 1 => empty()
 ~(zeroDim?(leq1,lvar)) => "failed"
leq1
package GUESS Guess

— Guess.input —

)set break resume
)sys rm -f Guess.output
)spool Guess.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Guess
--E 1

)spool
)lisp (bye)

— Guess.help —

====================================================================
Guess examples
====================================================================

The packages defined in this file enable {Axiom} to guess formulas for
sequences of, for example, rational numbers or rational functions,
given the first few terms. It extends and complements Christian
Krattenthaler’s program Rate and the relevant parts of Bruno Salvy and
Paul Zimmermann’s GFUN.

This package implements guessing of sequences. Packages for the
most common cases are provided as GuessInteger, GuessPolynomial, etc.

See Also:
 o )show Guess
Guess (GUESS)

Exports:

- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

The original code would not compile. This is a temporary replacement with changes marked with my initials. I will pick up the latest version sometime in the future and hope it compiles.
(Tim Daly)

— package GUESS Guess —

)abbrev package GUESS Guess
++ Author: Martin Rubey, Timothy Daly
++ Description:
++ This package implements guessing of sequences. Packages for the
++ most common cases are provided as \spadtype{GuessInteger},
++ \spadtype{GuessPolynomial}, etc.

Guess(F, S, EXPRR, R, retract, coerce): Exports == Implementation where
F: Field  -- zB.: FRAC POLY PF 5
-- in F we interpolate und check

S: GcdDomain

-- in guessExpRat I would like to determine the roots of polynomials in F. When
-- F is a quotientfield, I can get rid of the denominator. In this case F is
-- roughly QFCAT S

R: Join(OrderedSet, IntegralDomain)  -- zB.: FRAC POLY INT

-- results are given as elements of EXPRR
-- EXPRR: Join(ExpressionSpace, IntegralDomain,
  EXPRR: Join(FunctionSpace Integer, IntegralDomain,
    RetractableTo R, RetractableTo Symbol,
    RetractableTo Integer, CombinatorialOpsCategory,
    PartialDifferentialRing Symbol) with
PACKAGE GUESS GUESS

_*_ : (%,%) -> %
_/ : (%,%) -> %
_*_* : (%,%) -> %
numerator : % -> %
denominator : % -> %
ground? : % -> Boolean

-- zB.: EXPR INT
-- EXPR FRAC POLY INT is forbidden. Thus i cannot just use EXPR R
-- EXPRR exists, in case at some point there is support for EXPR PF 5.

-- the following I really would like to get rid of

retract: R -> F -- zB.: i+->i
coerce: F -> EXPRR -- zB.: i+->i
-- attention: EXPRR ~= EXPR R

LGOPT ==> List GuessOption
GOPTO ==> GuessOptionFunctions0

NNI ==> NonNegativeInteger
PI ==> PositiveInteger
EXPRI ==> Expression Integer
GUESSRESULT ==> List Record(function: EXPRR, order: NNI)

UFPSSF ==> UnivariateFormalPowerSeries F
UFPSS1 ==> UnivariateFormalPowerSeriesFunctions

UFPSS ==> UnivariateFormalPowerSeries S

SUP ==> SparseUnivariatePolynomial
UFPSSUF ==> UnivariateFormalPowerSeries SUP F

FFFG ==> FractionFreeFastGaussian
FFGF ==> FractionFreeFastGaussianFractions

-- CoeffAction
DIFFSPECA ==> (NNI, NNI, SUP S) -> S
DIFFSPECAF ==> (NNI, NNI, UFPSSUF) -> SUP F
DIFFSPECAX ==> (NNI, Symbol, EXPRR) -> EXPRR

-- the diagonal of the C-matrix
DIFFSPECC ==> NNI -> List S
HPSPEC ==> Record(guessStream: UFPSF -> Stream UFPSF,
degreeStream: Stream NNI,
testStream: UFPSSUPF -> Stream UFPSSUPF,
exprStream: (EXPRR, Symbol) -> Stream EXPRR,
A: DIFFSPECA,
AF: DIFFSPECAF,
AX: DIFFSPECAS,
C: DIFFSPECC)

-- note that empty?(guessStream.o) has to return always. In other words, if the
-- stream is finite, empty? should recognize it.

DIFFSPECN ==> EXPRR -> EXPRR -- eg.: i+->q^i

GUESSER ==> (List F, LGOPT) -> GUESSRESULT

FSUPS ==> Fraction SUP S
FSUPF ==> Fraction SUP F

V ==> OrderedVariableList(['a1, 'A])
POLYF ==> SparseMultivariatePolynomial(F, V)
FPOLYF ==> Fraction POLYF
FSUPFPOLYF ==> Fraction SUP FPOLYF
POLYS ==> SparseMultivariatePolynomial(S, V)
FPOLYS ==> Fraction POLYS
FSUPFPOLYS ==> Fraction SUP FPOLYS

--@<<implementation: Guess - Hermite-Pade - Types for Operators>>
-- EXT ==> (Integer, V, V) -> FPOLYS
-- EXTEXPR ==> (Symbol, F, F) -> EXPRR
Exports == with

guess: List F -> GUESSRESULT
  ++ \spad{guess l} applies recursively \spadfun{guessRec} and
  ++ \spadfun{guessADE} to the successive differences and quotients of
  ++ the list. Default options as described in
  ++ \spadtype{GuessOptionFunctions0} are used.

guess: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guess(l, options)} applies recursively \spadfun{guessRec}
  ++ and \spadfun{guessADE} to the successive differences and quotients
  ++ of the list. The given options are used.

guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
  ++ \spad{guess(l, guessers, ops)} applies recursively the given
  ++ guessers to the successive differences if ops contains the symbol
  ++ guessSum and quotients if ops contains the symbol guessProduct to
  ++ the list. Default options as described in
  ++ \spadtype{GuessOptionFunctions0} are used.
guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ \spad{guessSum} and quotients if ops contains the symbol
++ \spad{guessProduct} to the list. The given options are used.

guessExpRat: List F -> GUESSRESULT
++ \spad{guessExpRat l} tries to find a function of the form
++ n+->(a+b n)^n r(n), where r(n) is a rational function, that fits
++ l.

guessExpRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessExpRat(l, options)} tries to find a function of the
++ form n+->(a+b n)^n r(n), where r(n) is a rational function, that
++ fits l.

guessBinRat: List F -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
++ function, that fits l.

guessBinRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
++ function, that fits l.

if F has RetractableTo Symbol and S has RetractableTo Symbol then

guessExpRat: Symbol -> GUESSER
++ \spad{guessExpRat q} returns a guesser that tries to find a
++ function of the form n+->(a+b q^n)^n r(q^n), where r(q^n) is a
++ q-rational function, that fits l.

guessBinRat: Symbol -> GUESSER
++ \spad{guessBinRat q} returns a guesser that tries to find a
++ function of the form n+->qbinomial(a+b n, n) r(n), where r(q^n) is a
++ q-rational function, that fits l.

guessHP: (LGOPT -> HPSPEC) -> GUESSER
++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
++ approximation to the series generated by the given function f.

guessADE: List F -> GUESSRESULT
++ \spad{guessADE l} tries to find an algebraic differential equation
++ for a generating function whose first Taylor coefficients are
++ given by l, using the default options described in
++ \spad{type{GuessOptionFunctions0}}.

guessADE: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessADE(l, options)} tries to find an algebraic
++ differential equation for a generating function whose first Taylor
++ coefficients are given by 1, using the given options.

guessAlg: List F -> GUESSRESULT
++ \spad{guessAlg l} tries to find an algebraic equation for a
++ generating function whose first Taylor coefficients are given by
++ 1, using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessADE}(l, maxDerivative == 0).

guessAlg: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessAlg(l, options)} tries to find an algebraic equation
++ for a generating function whose first Taylor coefficients are
++ given by 1, using the given options. It is equivalent to
++ \spadfun{guessADE}(l, options) with \spad{maxDerivative == 0}.

guessHolo: List F -> GUESSRESULT
++ \spad{guessHolo l} tries to find an ordinary linear differential
++ equation for a generating function whose first Taylor coefficients
++ are given by 1, using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessADE}(l, maxPower == 1).

guessHolo: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessHolo(l, options)} tries to find an ordinary linear
differential equation for a generating function whose first Taylor
coefficients are given by 1, using the given options. It is
equivalent to \spadfun{guessADE}(l, options) with
++ \spad{maxPower == 1}.

guessPade: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessPade(l, options)} tries to find a rational function
++ whose first Taylor coefficients are given by 1, using the given
++ options. It is equivalent to \spadfun{guessADE}(l, maxDerivative == 0, maxPower == 1, allDegrees == true).

guessPade: List F -> GUESSRESULT
++ \spad{guessPade(l, options)} tries to find a rational function
++ whose first Taylor coefficients are given by 1, using the default
++ options described in \spadtype{GuessOptionFunctions0}. It is
equivalent to \spadfun{guessADE}(l, maxDerivative == 0, maxPower == 1, allDegrees == true).

guessRec: List F -> GUESSRESULT
++ \spad{guessRec l} tries to find an ordinary difference equation
++ whose first values are given by 1, using the default options
++ described in \spadtype{GuessOptionFunctions0}.

guessRec: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessRec(l, options)} tries to find an ordinary difference
++ equation whose first values are given by \( l \), using the given
++ options.

\texttt{guessPRec: (List \( F \), LGOPT) \rightarrow GUESSRESULT}
++ \texttt{\spad{\spad{\spad{\text{guessPRec}(l, options)}}}} tries to find a linear recurrence
++ with polynomial coefficients whose first values are given by \( l \),
++ using the given options. It is equivalent to
++ \texttt{\spad{\spad{\spad{\text{spadfun(guessRec)}\spad{(l, options)}}}} with \spad{\spad{\spad{\text{maxPower == 1}}}}}.

\texttt{guessPRec: List \( F \) \rightarrow GUESSRESULT}
++ \texttt{\spad{\spad{\spad{\text{guessPRec l}}}}} tries to find a linear recurrence with
++ polynomial coefficients whose first values are given by \( l \), using
++ the default options described in
++ \texttt{\spad{\spad{\spad{\text{\spadtype{GuessOptionFunctions0}}}}}}. It is equivalent to
++ \texttt{\spad{\spad{\spad{\text{spadfun(guessRec)}\spad{(l, maxPower == 1)}}}}}.

\texttt{guessRat: (List \( F \), LGOPT) \rightarrow GUESSRESULT}
++ \texttt{\spad{\spad{\spad{\text{guessRat}(l, options)}}}} tries to find a rational function
++ whose first values are given by \( l \), using the given options. It is
++ equivalent to \texttt{\spad{\spad{\spad{\text{spadfun(guessRec)}\spad{(l, maxShift == 0, maxPower
++ == 1, allDegrees == true)}}}}}.

\texttt{guessRat: List \( F \) \rightarrow GUESSRESULT}
++ \texttt{\spad{\spad{\spad{\text{guessRat l}}}}} tries to find a rational function whose first
++ values are given by \( l \), using the default options described in
++ \texttt{\spad{\spad{\spad{\text{\spadtype{GuessOptionFunctions0}}}}}}. It is equivalent to
++ \texttt{\spad{\spad{\spad{\text{spadfun(guessRec)}\spad{(l, maxShift == 0, maxPower == 1,
++ allDegrees == true)}}}}}.

\texttt{diffHP: LGOPT \rightarrow HPSPEC}
++ \texttt{\spad{\spad{\spad{\text{diffHP options}}}}} returns a specification for Hermite-Pade
++ approximation with the differential operator

\texttt{shiftHP: LGOPT \rightarrow HPSPEC}
++ \texttt{\spad{\spad{\spad{\text{shiftHP options}}}}} returns a specification for Hermite-Pade
++ approximation with the shift operator

if \( F \) has RetractableTo Symbol and \( S \) has RetractableTo Symbol then

\texttt{shiftHP: Symbol \rightarrow (LGOPT \rightarrow HPSPEC)}
++ \texttt{\spad{\spad{\spad{\text{shiftHP options}}}}} returns a specification for
++ Hermite-Pade approximation with the $q$-shift operator

\texttt{diffHP: Symbol \rightarrow (LGOPT \rightarrow HPSPEC)}
++ \texttt{\spad{\spad{\spad{\text{diffHP options}}}}} returns a specification for Hermite-Pade
++ approximation with the $q$-dilation operator

\texttt{guessRec: Symbol \rightarrow GUESSER}
++ \texttt{\spad{\spad{\spad{\text{guessRec q}}}}} returns a guesser that finds an ordinary
++ $q$-difference equation whose first values are given by \( l \), using
++ the given options.

guessPRec: Symbol -> GUESSER
++ \texttt{\spad{guessPRec q}} returns a guesser that tries to find
++ a linear q-recurrence with polynomial coefficients whose first
++ values are given by l, using the given options. It is
++ equivalent to \texttt{\spadfun{guessRec}\spad{(q)}} with
++ \texttt{\spad{maxPower == 1}}.

guessRat: Symbol -> GUESSER
++ \texttt{\spad{guessRat q}} returns a guesser that tries to find a
++ q-rational function whose first values are given by l, using
++ the given options. It is equivalent to \texttt{\spadfun{guessRec}} with
++ \texttt{\spad{(l, maxShift == 0, maxPower == 1, allDegrees == true)}}.

guessADE: Symbol -> GUESSER
++ \texttt{\spad{guessADE q}} returns a guesser that tries to find an
++ algebraic differential equation for a generating function whose
++ first Taylor coefficients are given by l, using the given
++ options.

-- We have to put this chunk at the beginning, because otherwise it will take
-- very long to compile.

ord1(x: List Integer, i: Integer): Integer ==
n := #x - 3 - i
x.(n+1)*reduce(_+, [x.j for j in 1..n], 0) + _
2*reduce(_+, [reduce(_+, [x.k*x.j for k in 1..j-1], 0) _
  for j in 1..n], 0)

ord2(x: List Integer, i: Integer): Integer ==
if zero? i then
  n := #x - 3 - i
  ord1(x, i) + reduce(_+, [x.j for j in 1..n], 0)*(x.(n+2)-x.(n+1))
else
  ord1(x, i)

deg1(x: List Integer, i: Integer): Integer ==
m := #x - 3
(x.(m+3)+x.(m+1)+x.(1+i))*reduce(_+, [x.j for j in 2+i..m], 0) + _
x.(m+3)*x.(m+1) + _
2*reduce(_+, [reduce(_+, [x.k*x.j for k in 2+i..j-1], 0) _
  for j in 2+i..m], 0)

deg2(x: List Integer, i: Integer): Integer ==
m := #x - 3
deg1(x, i) + _
\[(x.(m+3) + \text{reduce}(\_+, [x.j \text{ for } j \text{ in } 2+i..m], 0)) \times (x.(m+2)-x.(m+1))\]

for i in 1..1 by -1 repeat
den := eval(denominator res, n::EXPRR, (i-1)::EXPRR)
if den = 0 then return i::NNI
num := eval(numerator res, n::EXPRR, (i-1)::EXPRR)
if list.i ~= retract(retract(num/den)@R)
then return i::NNI
0$NNI

SUPS2SUPF(p: SUP S): SUP F ==
if F is S then
  p pretend SUP(F)
else if F is Fraction S then
  \text{map}(\text{coerce(#1)$Fraction(S)}, p)$SparseUnivariatePolynomialFunctions2(S, F)
else error "Type parameter F should be either equal to S or equal to Fraction S"

F2FPOLYS(p: F): FPOLYS ==
if F is S then
  p::POLYF::FPOLYF pretend FPOLYS
else if F is Fraction S then
  \text{numer}(p)$Fraction(S)::POLYS/denom(p)$Fraction(S)::POLYS
else error "Type parameter F should be either equal to S or equal to Fraction S"

MPCSF ==> MPolyCatFunctions2(V, IndexedExponents V, IndexedExponents V, S, F, POLYS, POLYF)

SUPF2EXPRR(xx: Symbol, p: SUP F): EXPRR ==
zero? p => 0
(coerce(leadingCoefficient p))::EXPRR \times (xx::EXPRR)**degree p + SUPF2EXPRR(xx, reductum p)

FSUPF2EXPRR(xx: Symbol, p: FSUPF): EXPRR ==
(SUPF2EXPRR(xx, numer p)) / (SUPF2EXPRR(xx, denom p))

POLYS2POLYF(p: POLYS): POLYF ==
if F is S then
  p pretend POLYF
else if F is Fraction S then
  \text{map}(\text{coerce(#1)$Fraction(S)}, p)$MPCSF
else error "Type parameter F should be either equal to S or equal -
SUPPOLYS2SUPF(p: SUP POLYS, a1v: F, Av: F): SUP F ==
  zero? p => 0
  lc: POLYF := POLYS2POLYF leadingCoefficient(p)
  monomial(retract(eval(lc, [index(1)$V, index(2)$V]::List V,
                      [a1v, Av]),
          degree p) + SUPPOLYS2SUPF(reductum p, a1v, Av)

SUPFPOLYS2SUPPOLYS(p: SUP FPOLYS): Fraction SUP POLYS ==
  cden := splitDenominator(p)
  $UnivariatePolynomialCommonDenominator(POLYS, FPOLYS,SUP FPOLYS)
  pnum: SUP POLYS := map(retract(#1 * cden.den)$FPOLYS, p)
  $SparseUnivariatePolynomialFunctions2(FPOLYS, POLYS)
  pden: SUP POLYS := (cden.den)::SUP POLYS
  pnum/pden

POLYF2EXPRR(p: POLYF): EXPRR ==
  map(convert(#1)@Symbol::EXPRR, coerce(#1)@EXPRR, p)
  $PolynomialCategoryLifting(IndexedExponents V, V,
                            F, POLYF, EXPRR)

-- this needs documentation. In particular, why is V appearing here?
GF ==> GeneralizedMultivariateFactorize(SingletonAsOrderedSet, IndexedExponents V, F, F, SUP F)

-- does not work:
--
-- WARNING (genufact): No known algorithm to factor ?, trying square-free.

-- GF ==> GenUFactorize F
defaultD: DIFFSPECN
defaultD(expr: EXPRR) := EXPRR == expr

-- applies n+->q^n or whatever DN is to i
DN2DL: (DIFFSPECN, Integer) -> F
DN2DL(DN, i) == retract(retract(DN(i::EXPRR))@R)

  res: List S := []
  d1 := degree(p1, val)
  d2 := degree(p2, val)
lead: $S$
for $k$ in $1..d-o+1$ repeat

$\text{p1atk := univariate eval(p1, vA, k::S)}$
$p2atk := \text{univariate eval(p2, vA, k::S)}$

$d1atk := \text{degree p1atk}$
$d2atk := \text{degree p2atk}$

-- output("k: " string(k))$OutputPackage
-- output("d1: " string(d1) " d1atk: " string(d1atk))$OutputPackage
-- output("d2: " string(d2) " d2atk: " string(d2atk))$OutputPackage

if $d2atk < d2$ then
  if $d1atk < d1$
    then lead := 0$S$
    else lead := $(\text{leadingCoefficient p1atk})^{((d2-d2atk)::\text{NNI})}$
  else
    if $d1atk < d1$
      then lead := $(-1S)^{d2} \times (\text{leadingCoefficient p2atk})^{((d1-d1atk)::\text{NNI})}$
    else lead := $1S$

if $\text{zero? lead}$
then res := cons(0, res)
else res := cons(lead \times (\text{resultant(p1atk, p2atk)})^{\text{SUP}(S)} \text{exquo } (k::S)^{(o::\text{NNI})})::S,$
res)

reverse res

$p(xm: \text{Integer}, i: \text{Integer}, va1: V, vA: V, \text{basis: DIFFSPECN})::\text{FPOLYS} = vA::\text{POLYS::FPOLYS} + va1::\text{POLYS::FPOLYS}$
$\times F2FPOLYS(DN2DL(\text{basis, i}) - DN2DL(\text{basis, xm}))$

$p2(xm: \text{Integer}, i: \text{Symbol, a1v: F, Av: F, basis: DIFFSPECN})::\text{EXPRR} =$
$\text{coerce(Av)} + \text{coerce(a1v)} \times (\text{basis(i::EXPRR)} - \text{basis(xm::EXPRR)})$

$\text{guessExpRatAux(xx: Symbol, list: List F, basis: DIFFSPECN,}$
$xValues: \text{List Integer, options: LGOPT})::\text{List EXPRR} =$

$a1: V := \text{index(1)}$V
$A: V := \text{index(2)}$V

len: NNI := #list
if len < 4 then return []
  else len := (len-3):NNI

xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]
$x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))$
$x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))$
\[ x_3 := \text{F2FPOLYS DN2DL(basis, xValues.(len+3))} \]

\[ y : \text{NNI \to FPOLYS :=} \]
\[ \text{F2FPOLYS(list.#1) \times -} \]
\[ \text{p(last xValues, (xValues.#1)::Integer, a1, A, basis)**-} \]
\[ (-xValues.#1)::Integer) \]

\[ \text{ylist: List FPOLYS := [y i for i in 1..len]} \]

\[ y_1 := y(len+1) \]
\[ y_2 := y(len+2) \]
\[ y_3 := y(len+3) \]

\[ \text{res := []::List EXPRR} \]
\[ \text{-- tpd: this is undefined since maxDegree is always nonnegative} \]
\[ \text{-- if maxDegree(options)$GOPT0 = -1} \]
\[ \text{then maxDeg := len-1} \]
\[ \text{else maxDeg := min(maxDegree(options)$GOPT0, len-1)} \]
\[ \text{maxDeg:Integer := min(tpd,len-1)} \]

\[ \text{for i in 0..maxDeg repeat} \]
\[ \text{if debug(options)$GOPT0 then} \]
\[ \text{output(hconcat("degree ExpRat ":OutputForm, i::OutputForm))} \]
\[ \text{OutputPackage} \]

\[ \text{if debug(options)$GOPT0 then} \]
\[ \text{systemCommand("sys date +%s")$MoreSystemCommands} \]
\[ \text{output("interpolating...":OutputForm)$OutputPackage} \]

\[ \text{ri: FSUPFPOLYS} \]
\[ := \text{interpolate(xlist, ylist, (len-1-i)::NNI) \times} \]
\[ \text{FFFG(FPOLYS, SUP FPOLYS)} \]

\[ \text{-- for experimental fraction free interpolation} \]
\[ \text{-- ri: Fraction SUP POLYS} \]
\[ := \text{interpolate(xlist, ylist, (len-1-i)::NNI) \times} \]
\[ \text{FFFG(POLYS, SUP POLYS)} \]

\[ \text{if debug(options)$GOPT0 then} \]
\[ \text{output(hconcat("xlist: ", xlist::OutputForm))$OutputPackage} \]
\[ \text{output(hconcat("ylist: ", ylist::OutputForm))$OutputPackage} \]
\[ \text{output(hconcat("ri: ", ri::OutputForm))$OutputPackage} \]
\[ \text{systemCommand("sys date +%s")$MoreSystemCommands} \]
\[ \text{output("polynomials...":OutputForm)$OutputPackage} \]

\[ \text{poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)} \]
\[ \text{poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)} \]
\[ \text{poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)} \]
-- for experimental fraction free interpolation
-- r2: FUPFFPOLYS := map(#1::FPOLYS, numer ri) 
-- $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)_
-- /map(#1::FPOLYS, denom ri) _
-- $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)
--
-- poly1: POLYS := numer(elt(ri2, x1)$SUP(FPOLYS) - y1)
-- poly2: POLYS := numer(elt(ri2, x2)$SUP(FPOLYS) - y2)
-- poly3: POLYS := numer(elt(ri2, x3)$SUP(FPOLYS) - y3)

n: Integer := len - i
o1: Integer := ord1(xValues, i)
d1: Integer := deg1(xValues, i)
o2: Integer := ord2(xValues, i)
d2: Integer := deg2(xValues, i)

-- another compiler bug: using i as iterator here makes the loop break

if debug(options)$GOPT0 then
  systemCommand("sys date +%s")$MoreSystemCommands
  output("interpolating resultants...":OutputForm)$OutputPackage

res1: SUP S
  := newton(evalResultant(poly1, poly3, o1, d1, a1, A))
  $NewtonInterpolation(S)

res2: SUP S
  := newton(evalResultant(poly2, poly3, o2, d2, a1, A))
  $NewtonInterpolation(S)

if debug(options)$GOPT0 then
  -- res1: SUP S := univariate(resultant(poly1, poly3, a1))
  -- res2: SUP S := univariate(resultant(poly2, poly3, a1))
  -- if res1~res1res or res2~res2res then
  --   output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
  --   output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
  --   output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
  --   output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  --   output(hconcat("res2 ", res2::OutputForm))$OutputPackage
  --   output("n/i: " string(n) " " string(i))$OutputPackage
  --   output("res1 ord: " string(o1) " " string(minimumDegree res1))
  --   $OutputPackage
  --   output("res1 deg: " string(d1) " " string(degree res1))
  --   $OutputPackage
  --   output("res2 ord: " string(o2) " " string(minimumDegree res2))
  --   $OutputPackage
  --   output("res2 deg: " string(d2) " " string(degree res2))
  --   $OutputPackage
if debug(options) $\& \text{GOPT0}$ then
    systemCommand("sys date +%s")$\& \text{MoreSystemCommands}
    output("computing gcd..."::OutputForm)$\& \text{OutputPackage}

-- we want to solve over $F$
-- for polynomial domains $S$ this seems to be very costly!
res3: SUP $F :=$ SUPS2SUPF(primitivePart(gcd(res1, res2)))

if debug(options) $\& \text{GOPT0}$ then
    systemCommand("sys date +%s")$\& \text{MoreSystemCommands}
    output("solving..."::OutputForm)$\& \text{OutputPackage}

-- res3 is a polynomial in $A=a0+(\text{len}+3)*a1$
-- now we have to find the roots of res3

for f in factors factor(res3)$\& \text{GF} \mid \text{degree f.factor} = 1$ repeat
    -- we are only interested in the linear factors
    if debug(options) $\& \text{GOPT0}$ then
        output(hconcat("f: ", f::OutputForm))$\& \text{OutputPackage}
        Av: F := \text{-coefficient}(f.factor, 0)
        / \text{leadingCoefficient} f.factor

-- FIXME: in an earlier version, we disregarded vanishing $Av$
-- maybe we intended to disregard vanishing $a1v$? Either doesn't really
-- make sense to me right now.

    evalPoly := eval(POLYS2POLYF poly3, A, Av)
    if zero? evalPoly
    then evalPoly := eval(POLYS2POLYF poly1, A, Av)
-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([1,1,1,1]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

for g in factors factor(univariate evalPoly)$\& \text{GF} \mid \text{degree g.factor} = 1$ repeat
    if debug(options) $\& \text{GOPT0}$ then
        output(hconcat("g: ", g::OutputForm))$\& \text{OutputPackage}
        a1v: F := \text{-coefficient}(g.factor, 0)
        / \text{leadingCoefficient} g.factor

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.

t1 := eval(POLYS2POLYF poly1, [a1, A]::List V,
            [a1v, Av]::List F)
if zero? t1 then
   t2 := eval(POLYS2POLYF poly2, [a1, A]::List V,
             [alv, Av]::List F)
if zero? t2 then

ri1: Fraction SUP POLYS
    := SUPFPOLYS2FSUPPOLYS(numer ri)
    / SUPFPOLYS2FSUPPOLYS(denom ri)

-- for experimental fraction free interpolation
--
ri1: Fraction SUP POLYS := ri

numr: SUP F := SUPPOLYS2SUPF(numer ri1, alv, Av)
denr: SUP F := SUPPOLYS2SUPF(denom ri1, alv, Av)

if not zero? denr then
   res4: EXPRR := eval(FSUPF2EXPRR(xx, numr/denr),
                        kernel(xx),
                        basis(xx::EXPRR))
   *p2(last xValues, _
       xx, alv, Av, basis)_
   **xx::EXPRR
   res := cons(res4, res)
else if zero? numr and debug(options)$GOPT0 then
   output("numerator and denominator vanish!")
$OutputPackage

-- If we are only interested in one solution, we do not try other degrees if we
-- have found already some solutions. I.e., the indentation here is correct.

if not null(res) and one(options)$GOPT0 then return res
res

guessExpRatAux0(list: List F, basis: DIFFSPECN, options: LGOPT): GUESSRESULT ==
if zero? safety(options)$GOPT0 then
   error "Guess: guessExpRat does not support zero safety"
-- guesses Functions of the Form (a1*n+a0)^n*rat(n)
xx := indexName(options)$GOPT0
-- restrict to safety
len: Integer := #list
if len-safety(options)$GOPT0+1 < 0 then return []
shortlist: List F := first(list, (len-safety(options)$GOPT0+1)::NNI)
-- remove zeros from list
zeros: EXPRR := 1
newlist: List F
xValues: List Integer

i: Integer := -1
for x in shortlist repeat
  i := i+1
  if x = 0 then
    zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))
  else
    i := -1
for x in shortlist repeat
  i := i+1
  if x ~= 0 then
    newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR, 
      i::EXPRR))@R),
    newlist)
    xValues := cons(i, xValues)

newlist := reverse newlist
xValues := reverse xValues

res: List EXPRR := [eval(zeros * f, xx::EXPRR, xx::EXPRR) 
  for f in guessExpRatAux(xx, newlist, basis, xValues, options)]
reslist := map(#1, checkResult(#1, xx, len, list, options], res)
$ListFunctions2(EXPRR, Record(function: EXPRR, order: NNI))

select(#1.order < len-safety(options)$GOPT0, reslist)

if F has RetractableTo Symbol and S has RetractableTo Symbol then
  guessExpRat(q: Symbol): GUESSER ==
  guessExpRatAux0(#1, q::EXPRR**#1, #2)

EXT ==> (Integer, V, V) -> FPOLYS
EXTEXPR ==> (Symbol, F, F) -> EXPRR

binExt: EXT
binExt(i: Integer, va1: V, vA: V): FPOLYS ==
  numl: List POLYS := [(vA::POLYS) + i * (va1::POLYS) - (l::POLYS) 
    for l in 0..i-1]
  num: POLYS := reduce(_, numl, 1)
num/(factorial(i)::POLYS)

binExtEXPR: EXTEXPR
binExtEXPR(i: Symbol, a1v: F, Av: F): EXPRR ==
  binomial(coerce Av + coerce a1v * (i::EXPRR), i::EXPRR)


  a1: V := index(1)$V
  A: V := index(2)$V

  len: NNI := #list
  if len < 4 then return []
  else len := (len-3)::NNI

  xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]
x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))
x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))
x3 := F2FPOLYS DN2DL(basis, xValues.(len+3))

  y: NNI -> FPOLYS :=
    F2FPOLYS(list.#1) / _
    ext((xValues.#1)::Integer, a1, A)

  ylist: List FPOLYS := [y i for i in 1..len]

  y1 := y(len+1)
y2 := y(len+2)
y3 := y(len+3)

  res := []::List EXPRR
  -- tpd: this is undefined since maxDegree is always nonnegative
  -- if maxDegree(options)$GOPT0 = -1
  -- then maxDeg := len-1
  -- else maxDeg := min(maxDegree(options)$GOPT0, len-1)
  -- maxDeg := min(maxDegree(options)$GOPT0, len-1)
tpd:Integer := (maxDegree(options)$GOPT0)::NNI::Integer
  maxDeg:Integer:=min(tpd,len-1)
  -- maxDeg:Integer := (maxDegree(options)$GOPT0)::NNI::Integer

  for i in 0..maxDeg repeat
    -- if debug(options)$GOPT0 then
    --   output(hconcat("degree BinRat ":OutputForm, i::OutputForm))
    --   $OutputPackage
    -- if debug(options)$GOPT0 then
ri: FSUPFPOLYS
:= interpolate(xlist, ylist, (len-1-i)::NNI) _
$FFFG(FPOLYS, SUP FPOLYS)

-- if debug(options)$GOPT0 then
-- output(hconcat("ri ", ri::OutputForm))$OutputPackage

poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)
poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)
poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)

-- if debug(options)$GOPT0 then
-- output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
-- output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
-- output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage

n:Integer := len - i
res1: SUP S := univariate(resultant(poly1, poly3, a1))
res2: SUP S := univariate(resultant(poly2, poly3, a1))
if debug(options)$GOPT0 then
  -- output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  -- output(hconcat("res2 ", res2::OutputForm))$OutputPackage

  -- if res1 ~ res1res or res2 ~ res2res then
  -- output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
  -- output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
  -- output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
  -- output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  -- output(hconcat("res2 ", res2::OutputForm))$OutputPackage

  output("n/i: " string(n) " " string(i))$OutputPackage
  output("res1 ord: " string(minimumDegree res1))
  $OutputPackage
  output("res1 deg: " string(degree res1))
  $OutputPackage
  output("res2 ord: " string(minimumDegree res2))
  $OutputPackage
  output("res2 deg: " string(degree res2))
  $OutputPackage

if debug(options)$GOPT0 then
  output("computing gcd...":OutputForm)$OutputPackage

-- we want to solve over F
  res3: SUP F := SUPS2SUPF(primitivePart(gcd(res1, res2)))

-- if debug(options)$GOPT0 then
-- output(hconcat("res3 ", res3::OutputForm))$OutputPackage
-- res3 is a polynomial in A=a0+(len+3)*a1
-- now we have to find the roots of res3

for f in factors factor(res3)$GF | degree f.factor = 1 repeat
-- we are only interested in the linear factors
-- if debug(options)$GOPT0 then
-- output(hconcat("f: ", f::OutputForm))$OutputPackage

Av: F := -coefficient(f.factor, 0)
   / leadingCoefficient f.factor
-- if debug(options)$GOPT0 then
-- output(hconcat("Av: ", Av::OutputForm))$OutputPackage

-- FIXME: in an earlier version, we disregarded vanishing Av
-- maybe we intended to disregard vanishing a1v? Either doesn't really
-- make sense to me right now.

evalPoly := eval(POLYS2POLYF poly3, A, Av)
if zero? evalPoly
then evalPoly := eval(POLYS2POLYF poly1, A, Av)
-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([1,1,1,1]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

for g in factors factor(univariate evalPoly)$GF
   | degree g.factor = 1 repeat
-- if debug(options)$GOPT0 then
-- output(hconcat("g: ", g::OutputForm))$OutputPackage

a1v: F := -coefficient(g.factor, 0)
   / leadingCoefficient g.factor
-- if debug(options)$GOPT0 then
-- output(hconcat("a1v: ", a1v::OutputForm))$OutputPackage

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.

t1 := eval(POLYS2POLYF poly1, [a1, A]::List V,
   [a1v, Av]::List F)

-- if debug(options)$GOPT0 then
-- output(hconcat("t1: ", t1::OutputForm))$OutputPackage

if zero? t1 then
t2 := eval(POLYS2POLYF poly2, [a1, A]::List V, [a1v, Av]::List F)

if debug(options)$GOPT0 then
  output(hconcat("t2: ", t2::OutputForm))$OutputPackage

if zero? t2 then

  rii: Fraction SUP POLYS
  := SUPFPOLYS2FSUPPOLYS(numer ri)
  / SUPFPOLYS2FSUPPOLYS(denom ri)

  if debug(options)$GOPT0 then
    output(hconcat("rii: ", rii::OutputForm))$OutputPackage

  numr: SUP F := SUPPOLYS2SUPF(numer rii, a1v, Av)
  denr: SUP F := SUPPOLYS2SUPF(denom rii, a1v, Av)

  if debug(options)$GOPT0 then
    output(hconcat("numr: ", numr::OutputForm))$OutputPackage
    output(hconcat("denr: ", denr::OutputForm))$OutputPackage

  if not zero? denr then
    res4: EXPRR := eval(FSUPF2EXPRR(xx, numr/denr),
                        kernel(xx),
                        basis(xx::EXPRR))
    * extEXPR(xx, a1v, Av)

    if debug(options)$GOPT0 then
      output(hconcat("res4: ", res4::OutputForm))$OutputPackage

    res := cons(res4, res)

  else if zero? numr and debug(options)$GOPT0 then
    output("numerator and denominator vanish!")$OutputPackage

  -- If we are only interested in one solution, we do not try other degrees if we
  -- have found already some solutions. I.e., the indentation here is correct.
  if not null(res) and one(options)$GOPT0 then return res

  res

guessBinRatAux0(list: List F, basis: DIFFSPECN, ext: EXT, extEXPR: EXTEXPR, options: LGOPT): GUESSRESULT ==

  if zero? safety(options)$GOPT0 then
    error "Guess: guessBinRat does not support zero safety"

  -- guesses Functions of the form binomial(a+b*n, n)*rat(n)
xx := indexName(options)$GOPT0

-- restrict to safety

len: Integer := #list
if len-safety(options)$GOPT0+1 < 0 then return []

shortlist: List F := first(list, (len-safety(options)$GOPT0+1)::NNI)

-- remove zeros from list

zeros: EXPRR := 1
newlist: List F
xValues: List Integer

i: Integer := -1
for x in shortlist repeat
  i := i+1
  if x = 0 then
    zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))
  i := -1
for x in shortlist repeat
  i := i+1
  if x ~= 0 then
    newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR, i::EXPRR))@R),
                    newlist)
    xValues := cons(i, xValues)
newlist := reverse newlist
xValues := reverse xValues

res: List EXPRR
  := [eval(zeros * f, xx::EXPRR, xx::EXPRR) _
      for f in guessBinRatAux(xx, newlist, basis, ext, extEXPR, xValues, _
                              options)]
reslist := map([#1, checkResult(#1, xx, len, list, options)], res)
$ListFunctions2(EXPRR, Record(function: EXPRR, order: NNI))

select(#1.order < len-safety(options)$GOPT0, reslist)

guessBinRat(list : List F): GUESSRESULT ==
guessBinRatAux0(list, defaultD, binExt, binExtEXPR, [])

guessBinRat(list: List F, options: LGOPT): GUESSRESULT ==
guessBinRatAux0(list, defaultD, binExt, binExtEXPR, options)
if $F$ has RetractableTo Symbol and $S$ has RetractableTo Symbol then

$qD: \text{Symbol} \rightarrow \text{DIFFSPECN}$
$qD \ q \ := \ (q::\text{EXPRR})^{*\#1}$

$q\text{BinExtAux}(q: \text{Symbol}, i: \text{Integer}, vA1: \text{V}, vA: \text{V}): \text{FPOLYS} \equiv$

$f1: \text{List} \ \text{FPOLYS}$
$:= \ [(1$FPOLYS \ -$ \ va1::\text{POLYS}::\text{FPOLYS} \ * \ (vA::\text{POLYS}::\text{FPOLYS})^{*(i-1)} \ * \$
$F2FPOLYS(q::F)^{*1}) \ / \ (1-F2FPOLYS(q::F)^{*1}) \ _$
$\text{for} \ l \ in \ 1..i]$
$\text{reduce}(_{\ast}, \ f1, 1)$

$q\text{BinExt}: \text{Symbol} \rightarrow \text{EXT}$
$q\text{BinExt} \ q \ := \ q\text{BinExtAux}(q, \ #1, \ #2, \ #3)$

$q\text{BinExtEXPRAux}(q: \text{Symbol}, i: \text{Symbol}, a1v: \text{F}, Av: \text{F}): \text{EXPRR} \equiv$

$l: \text{Symbol} := \ 'l$
$product((1$EXPRR \ -$\ _$
$\text{coerce} \ a1v \ * \ (\text{coerce} \ Av) \ ** \ (\text{coerce} \ i - 1$EXPRR) \ * \ _$
$(q::EXPRR) ** (\text{coerce}(1)) / _$
$(1$EXPRR - (q::EXPRR) ** (\text{coerce}(1)), _$
$\text{equation}(1, 1$EXPRR..i::EXPRR))$

$q\text{BinExtEXPR}: \text{Symbol} \rightarrow \text{EXTEXT}$
$q\text{BinExtEXPR} \ q \ := \ q\text{BinExtEXPRAux}(q, \ #1, \ #2, \ #3)$

guessBinRat(q: \text{Symbol}): \text{GUESSER} \equiv$

guessBinRatAux0(#1, qD \ q, q\text{BinExt} \ q, q\text{BinExtEXPR} \ q, \ #2)_

-- some useful types for Ore operators that work on series

-- the differentiation operator
 DIFFSPECX ==> (EXPRR, Symbol, NonNegativeInteger) \rightarrow \ EXPRR
 -- eg.: $f(x)+\rightarrow f(q\cdot x)$
 -- $f(x)+\rightarrow D(f, x)$
 DIFFSPECX ==> (UFPSSF, NonNegativeInteger) \rightarrow \ UFPSSF
 -- eg.: $f(x)+\rightarrow f(q\cdot x)$
 DIFFSPECSSF ==> (UFPSSUPF, NonNegativeInteger) \rightarrow \ UFPSSUPF
 -- eg.: $f(x)+\rightarrow f(q\cdot x)$

-- the constant term for the inhomogeneous case
 DIFFSPEC1 ==> UFPSSF
 DIFFSPEC1F ==> UFPSSUPF
 DIFFSPEC1X ==> Symbol \rightarrow \ EXPRR
termAsEXPRR(f: EXPRR, xx: Symbol, l: List Integer, 
   D1X: DIFFSPEC1X): EXPRR == 
   if empty? l then D1X(xx) 
   else 
       ll: List List Integer := powers(l)$Partition 
       f1: List EXPRR := [DX(f, xx, (first part-1)::NonNegativeInteger) 
           ** second(part)::NNI for part in ll]
       reduce(_, f1)

termAsUFPSF(f: UFPSF, l: List Integer, DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF == 
   if empty? l then D1 
   else 
       ll: List List Integer := powers(l)$Partition 
       -- first of each element of ll is the derivative, second is the power 
       f1: List UFPSF := [DS(f, (first part -1)::NonNegativeInteger) _ 
           ** second(part)::NNI for part in ll]
       reduce(_, f1)

-- returns \prod f^(l.i), but using the Hadamard product
termAsUFPSF2(f: UFPSF, l: List Integer, 
   DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF == 
   if empty? l then D1 
   else 
       ll: List List Integer := powers(l)$Partition 
       -- first of each element of ll is the derivative, second is the power 
       f1: List UFPSF 
           := [map(#1** second(part)::NNI, DS(f, (first part -1)::NNI)) _ 
               for part in ll]
       reduce(hadamard$UFPS1(F), f1)

termAsUFPSSUPF(f: UFPSSUPF, l: List Integer, 
   DSF: DIFFSPECSF, D1F: DIFFSPEC1F): UFPSSUPF == 
   if empty? l then D1F 
   else 
       ll: List List Integer := powers(l)$Partition 
       -- first of each element of ll is the derivative, second is the power 
       f1: List UFPSSUPF 
           := [DSF(f, (first part -1)::NonNegativeInteger) _ 
               ** second(part)::NNI for part in ll]
reduce(_, f)

-- returns $\prod f(l_.i)$, but using the Hadamard product
termAsUFPSSUFP2(f: UFPSSUFP, l: List Integer,
   DSF: DIFFSPEC2SF, D1F: DIFFSPEC1F): UFPSSUFP ==
   if empty? l then D1F
   else
      ll: List List Integer := powers(l)$Partition
      -- first of each element of ll is the derivative, second is the power
      fl: List UFPSSUFP
         := [map(#1 ** second(part)::NNI, DSF(f, (first part -1)::NNI))
            for part in ll]
      reduce(hadamard$UFPS1(SUP F), fl)

FilteredPartitionStream(options: LGOPT): Stream List Integer ==
-- tpd: must force types to NNI and Integer
maxD:Integer := 1+maxDerivative(options)$GOPT0::NNI::Integer
maxP:Integer := maxPower(options)$GOPT0::NNI::Integer
if maxD > 0 and maxP > -1 then
   s := partitions(maxD, maxP)$PartitionsAndPermutations
else
   s1: Stream Integer := generate(inc, 1)$Stream(Integer)
   s2: Stream Stream List Integer
      := map(partitions(#1)$PartitionsAndPermutations, s1)
      $StreamFunctions2(Integer, Stream List Integer)
   s3: Stream List Integer
      := concat(s2)$StreamFunctions1(List Integer)
      -- s := cons([],
      -- select(((maxD = 0) or (first #1 <= maxD))
      --       and ((maxP = -1) or (# #1 <= maxP)), s3))
   s := cons([],
      select(((maxD = 0) or (#1 -> maxD))
      and ((maxP = -1) or (first #1 <= maxP)), s3))

   s := conjugates(s)$PartitionsAndPermutations
   -- tpd: force the Boolean branch
   tpd2:Boolean:=homogeneous(options)$GOPT0::Boolean
   if tpd2 then rest s else s
else
   -- for functions
   ADEguessStream(f: UFPFSF, partitions: Stream List Integer,
      DS: DIFFSPEC2, D1: DIFFSPEC1): Stream UFPSF ==
map(termAsUFPSF(f, #1, DS, D1), partitions)
$\text{StreamFunctions2}(\text{List Integer, UFPSF})$

-- for coefficients, i.e., using the Hadamard product
ADEguessStream2(f: UFPSF, partitions: Stream List Integer,
DS: DIFFSPECS, D1: DIFFSPEC1): Stream UFPSF ==
map(termAsUFPSF2(f, #1, DS, D1), partitions)
$\text{StreamFunctions2}(\text{List Integer, UFPSF})$

ADEdegreeStream(partitions: Stream List Integer): Stream NNI ==
scan(0, \max(\text{if empty?} #1 \text{ then } 0 \text{ else (first } #1 - 1)::\text{NNI}, #2),
partitions)$\text{StreamFunctions2}(\text{List Integer, NNI})$

ADEtestStream(f: UFPSSUPF, partitions: Stream List Integer,
DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
map(termAsUFPSSUPF(f, #1, DSF, D1F), partitions)
$\text{StreamFunctions2}(\text{List Integer, UFPSSUPF})$

ADEtestStream2(f: UFPSSUPF, partitions: Stream List Integer,
DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
map(termAsUFPSSUPF2(f, #1, DSF, D1F), partitions)
$\text{StreamFunctions2}(\text{List Integer, UFPSSUPF})$

ADEEXPRRStream(f: EXPRR, xx: Symbol, partitions: Stream List Integer,
DX: DIFFSPECX, D1X: DIFFSPEC1X): Stream EXPRR ==
map(termAsEXPRR(f, xx, #1, DX, D1X), partitions)
$\text{StreamFunctions2}(\text{List Integer, EXPRR})$

diffDX: DIFFSPECX
diffDX(expr, x, n) == D(expr, x, n)

diffDS: DIFFSPECS
diffDS(s, n) == D(s, n)

diffDSF: DIFFSPECSF
diffDSF(s, n) ==
-- I have to help the compiler here a little to choose the right signature...
  if SUP F has \_*: (NonNegativeInteger, SUP F) \to SUP F
  then D(s, n)
diffAX: DIFFSPECAX
diffAX(l: NNI, x: Symbol, f: EXPRR): EXPRR ==
  (x::EXPRR)**l * f
diffA: DIFFSPECA
diffA(k: NNI, l: NNI, f: SUP S): S ==
  DiffAction(k, l, f)$\text{FFFG}(S, SUP S)$

diffAF: DIFFSPECAF
diffAF(k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
\text{DiffAction}(x, 1, f)\text{FFFG}(\text{SUP F, UFPSSUPF})

\text{diffC: DIFFSPECC}
\text{diffC}(\text{total: NNI}): \text{List } S == \text{DiffC}(\text{total})\text{FFFG}(S, \text{SUP S})

\text{diff1X: DIFFSPEC1X}
\text{diff1X}(x: \text{Symbol}) == 1\text{EXPRR}

\text{diffHP options ==}
if displayAsGF(options)\text{GOPT0} then
  \text{partitions} := \text{FilteredPartitionStream options}
  \[\text{ADEguessStream(#1, partitions, diffDS, 1\text{UFPSF}), }\]
  \[\text{ADEdegreeStream partitions, }\]
  \[\text{ADEtestStream(#1, partitions, diffDSF, 1\text{UFPSSUPF}), }\]
  \[\text{ADEEXPRRStream(#1, #2, partitions, diffDX, diff1X), }\]
  \[\text{diffA, diffAF, diffAX, diffC}]\text{HPSPEC}
else
  \text{error "Guess: guessADE supports only displayAsGF"}

if F has RetractableTo Symbol and S has RetractableTo Symbol then

\text{qDiffDX(q: Symbol, expr: EXPRR, x: Symbol, n: NonNegativeInteger): EXPRR ==}
\text{eval(expr, x::EXPRR, (q::EXPRR)**n*x::EXPRR)}

\text{qDiffDS(q: Symbol, s: UFPSF, n: NonNegativeInteger): UFPSF ==}
\text{multiplyCoefficients((q::F)**((n*#1)::NonNegativeInteger), s)}

\text{qDiffDSF(q: Symbol, s: UFPSSUPF, n: NonNegativeInteger): UFPSSUPF ==}
\text{multiplyCoefficients((q::F::SUP F)**((n*#1)::NonNegativeInteger), s)}

\text{diffHP(q: Symbol): (LGOPT \rightarrow \text{HPSPEC}) ==}
if displayAsGF(#1)\text{GOPT0} then
  \text{partitions} := \text{FilteredPartitionStream #1}
  \[\text{ADEguessStream(#1, partitions, qDiffDS(q, #1, #2), 1\text{UFPSF}), }\]
  \[\text{repeating([0\text{NNI}])\text{Stream(NNI)}, }\]
  \[\text{ADEtestStream(#1, partitions, qDiffDSF(q, #1, #2), 1\text{UFPSSUPF}), }\]
  \[\text{ADEEXPRRStream(#1, #2, partitions, qDiffDX(q, #1, #2, #3), diff1X), }\]
  \[\text{diffA, diffAF, diffAX, diffC}]\text{HPSPEC}
else
  \text{error "Guess: guessADE supports only displayAsGF"}

\text{ShiftSX(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==}
\text{eval(expr, x::EXPRR, x::EXPRR+n::EXPRR)}

\text{ShiftSXGF(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==}
if zero? n then expr
else
  \[l := \text{eval(D(expr, x, i)/factorial(i)::EXPRR, x::EXPRR, 0\text{EXPRR})}\]
  \[*(x::EXPRR)**i \text{ for } i \text{ in } 0..\text{n-1}\]
  \[](expr\text{-reduce}(\text{+}, l))/(x::EXPRR**n)
ShiftSS(s:UFPSF, n:NNI): UFPSF ==
((quoByVar #1)**n)$MappingPackage1(UFPSF) (s)

ShiftSF(s:UFPSUPF, n: NNI):UFPSUPF ==
((quoByVar #1)**n)$MappingPackage1(UFPSUPF) (s)

ShiftAX(l: NNI, n: Symbol, f: EXPRR): EXPRR ==
n::EXPRR**l * f

ShiftAXGF(l: NNI, x: Symbol, f: EXPRR): EXPRR ==
-- I need to help the compiler here, unfortunately
if zero? l then f else
   s := [stirling2(l, i)$IntegerCombinatoricFunctions(Integer)::EXPRR *
      (x::EXPRR)**i*D(f, x, i) for i in 1..l]
    reduce(_+, s)

ShiftA(k: NNI, l: NNI, f: SUP S): S ==
ShiftAction(k, l, f)$FFFG(S, SUP S)

ShiftAF(k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
ShiftAction(k, l, f)$FFFG(SUP F, UFPSSUPF)

ShiftC(total: NNI): List S ==
ShiftC(total)$FFFG(S, SUP S)

shiftHP options ==
partitions := FilteredPartitionStream options
if displayAsGF(options)$GOPT0 then
   if maxPower(options)$GOPT0 = 1 then
      [ADEguessStream(#1, partitions, ShiftSS, (1-monomial(1,1))**(-1)), _,
       ADEdegreeStream partitions, _
       ADEtestStream(#1, partitions, ShiftSF, (1-monomial(1,1))**(-1)), _
       ADEEXPRRStream(#1, #2, partitions, ShiftSXGF, 1/(1-#1::EXPRR)), _
       ShiftA, ShiftAF, ShiftAXGF, ShiftC]$HPSPEC
   else
      error "Guess: no support for the Shift operator with displayAsGF _
      and maxPower>1"
   else
      [ADEguessStream2(#1, partitions, ShiftSS, (1-monomial(1,1))**(-1)), _,
       ADEdegreeStream partitions, _
       ADEtestStream2(#1, partitions, ShiftSF, (1-monomial(1,1))**(-1)), _
       ADEEXPRRStream2(#1, #2, partitions, ShiftSX, diff1X), _
       ShiftA, ShiftAF, ShiftAX, ShiftC]$HPSPEC
   end

if F has RetractableTo Symbol and S has RetractableTo Symbol then

qShiftAX(q: Symbol, l: NNI, n: Symbol, f: EXPRR): EXPRR ==
(q::EXPRR)**(l*n::EXPRR) * f
qShiftA(q: Symbol, k: NNI, l: NNI, f: SUP S): S ==
 qShiftAction(q::S, k, l, f)$FFFG(S, SUP S)

qShiftAF(q: Symbol, k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
 qShiftAction(q::F::SUP(F), k, l, f)$FFFG(SUP F, UFPSSUPF)

qShiftC(q: Symbol, total: NNI): List S ==
 qShiftC(q::S, total)$FFFG(S, SUP S)

shiftHP(q: Symbol): (LGOPT -> HPSPEC) ==
 partitions := FilteredPartitionStream #1
 if displayAsGF(#1)$GOPT0 then
  error "Guess: no support for the qShift operator with displayAsGF"
 else
  [ADEguessStream2(#1, partitions, ShiftSS, _
   (1-monomial(1,1))**(-1)), _
   ADEdegreeStream partitions, _
   ADEtestStream2(#1, partitions, ShiftSF, _
   (1-monomial(1,1))**(-1)), _
   ADEEXPRStream(#1, #2, partitions, ShiftSX, diff1X), _
   qShiftA(q, #1, #2, #3), qShiftAF(q, #1, #2, #3), _
   qShiftAX(q, #1, #2, #3), qShiftC(q, #1)]$HPSPEC

makeEXPRR(DAX: DIFFSPECAX, x: Symbol, p: SUP F, expr: EXPRR): EXPRR ==
 if zero? p then 0$EXPRR
 else
  coerce(leadingCoefficient p)::EXPRR * DAX(degree p, x, expr) _
  + makeEXPRR(DAX, x, reductum p, expr)

list2UFPSF(list: List F): UFPSF ==
 series(list::Stream F)$UFPSF

list2UFPSSUPF(list: List F): UFPSSUPF ==
  l := [e::SUP(F) for e in list for i in 0..]::Stream SUP F
  series(l)$UFPPSSUPF + monomial(monomial(1,1)$SUP(F), #list)$UFPPSSUPF

SUPF2SUPSUPF(p: SUP F): SUP SUP F ==
 map(#1::SUP F, p)$SparseUnivariatePolynomialFunctions2(F, SUP F)

UFPSF2SUPF(f: UFPSF, deg: NNI): SUP F ==
 makeSUP univariatePolynomial(f, deg)

getListSUPF(s: Stream UFPSF, o: NNI, deg: NNI): List SUP F ==
 map(UFPSF2SUPF(#1, deg), entries complete first(s, o))
 $ListFunctions2(UFPSF, SUP F)

S2EXPRR(s: S): EXPRR ==
 if F is S then
  coerce(s pretend F)$EXPRR
 else if F is Fraction S then
coerce(s::Fraction(S))@EXPRR
else error "Type parameter F should be either equal to S or equal to Fraction S"

guessInterpolate(guessList: List SUP F, eta: List NNI, D: HPSPEC)
: Matrix SUP S ==
if F is S then
  vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
  generalInterpolation((D.C)(reduce(_+, eta)), D.A, vguessList, eta)$FFFG(S, SUP S)
else if F is Fraction S then
  vguessListF: Vector SUP F := vector(guessList)
  generalInterpolation((D.C)(reduce(_+, eta)), D.A, vguessListF, eta)$FFFGF(S, SUP S, SUP F)
else error "Type parameter F should be either equal to S or equal to Fraction S"

guessInterpolate2(guessList: List SUP F, sumEta: NNI, maxEta: NNI, D: HPSPEC)
: Stream Matrix SUP S ==
if F is S then
  vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
  generalInterpolation((D.C)(sumEta), D.A, vguessList, sumEta, maxEta)
  $FFFG(S, SUP S)
else if F is Fraction S then
  vguessListF: Vector SUP F := vector(guessList)
  generalInterpolation((D.C)(sumEta), D.A, vguessListF, sumEta, maxEta)
  $FFFGF(S, SUP S, SUP F)
else error "Type parameter F should be either equal to S or equal to Fraction S"

: Union("failed", Record(function: EXPRR, order: NNI)) ==
--tpd: maxDegree is defined to be nonnegative
-- ((maxDegree(options)$GOPT0 = -1) and
((allDegrees(options)$GOPT0 = false) and
 zero?(last resi))
=> return "failed"
nonZeroCoefficient: Integer := 0
for i in 1..#resi repeat
    if not zero? resi.i then
        if zero? nonZeroCoefficient then
            nonZeroCoefficient := i
        else
            nonZeroCoefficient := 0
        break
    if not zero? nonZeroCoefficient then
        (freeOf?(exprList.nonZeroCoefficient, name op)) => return "failed"

for e in list repeat
    if not zero? e then return "failed"
else
    resiSUPF := map(SUPF2SUPSUPF SUPS2SUPF #1, resi)
    $ListFunctions2(SUP S, SUP SUP F)
    iterate? := true;
    for d in guessDegree+1.. repeat
        c: SUP F := generalCoefficient(D.AF, vector testList, 
d, vector resiSUPF) $FFFG(SUP F, UFPSSUPF)
        if not zero? c then
            iterate? := ground? c
            break
    iterate? => return "failed"

    g: SUP S
    if S has Field
        then g := leadingCoefficient(find(not zero? #1, reverse resi)::SUP(S))::SUP(S)
    else g := gcd resi
    resiF := map(SUPS2SUPF((#1 exquo g)::SUP(S)), resi) $ListFunctions2(SUP S, SUP F)

    if debug(options)$GOPT0 then
        output(hconcat("trying possible solution ", resiF::OutputForm)) $OutputPackage

    -- transform each term into an expression
    ex: List EXPRR := [makeEXPRR(D.AX, dummy, p, e) _
                        for p in resiF for e in exprList]

    -- transform the list of expressions into a sum of expressions
    res: EXPRR
    if displayAsGF(options)$GOPT0 then
        res := evalADE(op, dummy, variableName(options)$GOPT0::EXPRR, 
                        indexName(options)$GOPT0::EXPRR, 
                        indexName(options)$GOPT0::EXPRR...
numerator reduce(_+, ex),
reverse initials)
$\text{RecurrenceOperator}(\text{Integer}, \text{EXPRR})$
ord: \text{NNI} := 0

-- FIXME: checkResult doesn't really work yet for generating functions
else
res := evalRec(op, dummy, indexName(options)\$\text{GOPT0}\::\text{EXPRR},
indexName(options)\$\text{GOPT0}\::\text{EXPRR},
numerator reduce(_+, ex),
reverse initials)
$\text{RecurrenceOperator}(\text{Integer}, \text{EXPRR})$
ord: \text{NNI} := checkResult(res, indexName(options)\$\text{GOPT0}, _
#list, list, options)

[res, ord]\$\text{Record}(\text{function: EXPRR, order: NNI})
guessHPaux(list: \text{List} \text{F}, D: \text{HPSPEC}, \text{options: LGOPT}): \text{GUESSRESULT} ==
reslist: \text{GUESSRESULT} := []
listDegree := \#list-1\$\text{safety(options)\$\text{GOPT0}}
if listDegree < 0 then return reslist
a := functionName(options)\$\text{GOPT0}
op := operator a
x := variableName(options)\$\text{GOPT0}
dummy := new$\text{Symbol}
initials: \text{List EXPRR} := [coerce(e)@\text{EXPRR} for e in list]
guessS := (D.guessStream)(list2UFPSF list)
degreeS := D.degreeStream
testS := (D.testStream)(list2UFPSSUPF list)
exprS := (D.exprStream)(op(dummy::\text{EXPRR})::\text{EXPRR}, dummy)
iterate?: \text{Boolean} := false -- this is necessary because the compiler
-- doesn't understand => "iterate" properly
-- the latter just leaves the current block, it
-- seems
for o in 2.. repeat
empty? rest(guessS, (o-1)::\text{NNI}) => break
guessDegree: \text{Integer} := listDegree-(degreeS.o)::\text{Integer}
guessDegree < 0 => break
if debug(options)\$\text{GOPT0} then
output(hconcat("Trying order ", o::\text{OutputForm})\$\text{OutputPackage}
output(hconcat("guessDegree is ", guessDegree::\text{OutputForm}))
$\text{OutputPackage}$
if allDegrees(options)\$\text{GOPT0} then
(o > guessDegree+2) => return reslist
--tpd: force NNI and Integer
maxEta: \text{Integer} := 1+maxDegree(options)\$\text{GOPT0)::\text{NNI}::\text{Integer}
if maxEta = 0
then maxEta := guessDegree+1
else
    maxParams := divide(guessDegree::NNI+1, o)
    if debug(options)$GOPT0
        then output(hconcat("maxParams: ", maxParams::OutputForm))
        $OutputPackage
    if maxParams.quotient = 0 and maxParams.remainder < o-1
    then return reslist
--tpd: maxDegree is defined to be nonnegative
--
if ((maxDegree(options)$GOPT0 ~= -1) and
 if ((maxDegree(options)$GOPT0::NNI::Integer < maxParams.quotient)) and
 not (empty? rest(guessS, o) or
 ((newGuessDegree := listDegree-(degreeS.(o+1))::Integer)
      < 0) or
 ((newMaxParams := divide(newGuessDegree::NNI+1, o+1))
       .quotient = 0) and
 (newMaxParams.remainder < o)))
    then iterate? := true
--tpd: maxDegree is defined to be nonnegative
else if ((maxDegree(options)$GOPT0 ~= -1) and
 if (maxParams.quotient > maxDegree(options)$GOPT0::NNI::Integer)
    then
      --tpd: maxDegree is defined to be nonnegative
      guessDegree := o*(1+maxDegree(options)$GOPT0::NNI::Integer)-2
      eta: List NNI
          := [(if i < o then maxDegree(options)$GOPT0::NNI + 1 else maxDegree(options)$GOPT0::NNI)
              for i in 1..o]
      else eta: List NNI
          := [(if i <= maxParams.remainder then maxParams.quotient + 1 else maxParams.quotient)::NNI
              for i in 1..o]
    if iterate?
    then
      iterate? := false
      if debug(options)$GOPT0 then output("iterating")$OutputPackage
    else
        guessList: List SUP F := getListSUPF(guessS, o, guessDegree::NNI)
        testList: List UFPSSUPF := entries complete first(testS, o)
        exprList: List EXPRR := entries complete first(exprS, o)
        if debug(options)$GOPT0 then
            output("The list of expressions is")$OutputPackage
            output(exprList::OutputForm)$OutputPackage
        if allDegrees(options)$GOPT0 then
            MS: Stream Matrix SUP S := guessInterpolate2(guessList,
                                         guessDegree::NNI+1,
                                         maxEta::NNI, D)
repeat
  (empty? MS) \Rightarrow break
  M := first MS

  for i in 1..0 repeat
    res := testInterpolant(entries column(M, i),
                            list,
                            testList,
                            exprList,
                            initials,
                            guessDegree::NNI,
                            D, dummy, op, options)

    (res case "failed") \Rightarrow "iterate"

    if not member?(res, reslist)
    then reslist := cons(res, reslist)

    if one(options)$GOPT0 then return reslist

  MS := rest MS
else
  M := Matrix SUP S := guessInterpolate(guessList, eta, D)

  for i in 1..0 repeat
    res := testInterpolant(entries column(M, i),
                            list,
                            testList,
                            exprList,
                            initials,
                            guessDegree::NNI,
                            D, dummy, op, options)

    (res case "failed") \Rightarrow "iterate"

    if not member?(res, reslist)
    then reslist := cons(res, reslist)

    if one(options)$GOPT0 then return reslist

reslist

guessHP(D: LGOPT -> HPSPEC): GUESSER == guessHPaux(#1, D #2, #2)

--tpd comment out the call to displayAsGF. it won't type check
guessADE(list: List F, options: LGOPT): GUESSRESULT ==
--tpd
  options := LGOPT := cons(displayAsGF(true)$GuessOption, options)
  opts := options
guessHPaux(list, diffHP opts, opts)

guessADE(list: List F): GUESSRESULT == guessADE(list, [])
guessAlg(list: List F, options: LGOPT) ==
    guessADE(list, cons(maxDerivative(0)$GuessOption, options))

guessAlg(list: List F): GUESSRESULT == guessAlg(list, [])

guessHolo(list: List F, options: LGOPT): GUESSRESULT ==
    guessADE(list, cons(maxPower(1)$GuessOption, options))

guessHolo(list: List F): GUESSRESULT == guessHolo(list, [])

guessPade(list: List F, options: LGOPT): GUESSRESULT ==
    opts := append(options, [maxDerivative(0)$GuessOption,
                            maxPower(1)$GuessOption,
                            allDegrees(true)$GuessOption])
    guessADE(list, opts)

guessPade(list: List F): GUESSRESULT == guessPade(list, [])

if F has RetractableTo Symbol and S has RetractableTo Symbol then
    --tpd comment out the call to displayAsGF. it won't type check
    guessADE(q: Symbol): GUESSER ==
        --tpd opts: LGOPT := cons(displayAsGF(true)$GuessOption, #2)
        opts := #2
        guessHPaux(#1, (diffHP q)(opts), opts)

    --tpd comment out the call to displayAsGF. it won't type check
    guessRec(list: List F, options: LGOPT): GUESSRESULT ==
        --tpd opts: LGOPT := cons(displayAsGF(false)$GuessOption, options)
        opts := options
        guessHPaux(list, shiftHP opts, opts)

    guessRec(list: List F): GUESSRESULT == guessRec(list, [])

guessPRec(list: List F, options: LGOPT): GUESSRESULT ==
    guessRec(list, cons(maxPower(1)$GuessOption, options))

guessPRec(list: List F): GUESSRESULT == guessPRec(list, [])

guessRat(list: List F, options: LGOPT): GUESSRESULT ==
    opts := append(options, [maxShift(0)$GuessOption,
                            maxPower(1)$GuessOption,
                            allDegrees(true)$GuessOption])
    guessRec(list, opts)

guessRat(list: List F): GUESSRESULT == guessRat(list, [])

if F has RetractableTo Symbol and S has RetractableTo Symbol then
--tpd comment out the call to displayAsGF. it won't type check

guessRec(q: Symbol): GUESSER ==
-- opts: LGOPT := cons(displayAsGF(false)$GuessOption, #2)
  opts := #2
guessHPaux(#1, (shiftHP q)(opts), opts)

--tpd comment out the call to displayAsGF. it won't type check

guessPRec(q: Symbol): GUESSER ==
-- opts: LGOPT := append([displayAsGF(false)$GuessOption, maxPower(1)$GuessOption], #2)
  opts := #2
guessHPaux(#1, (shiftHP q)(opts), opts)

--tpd comment out the call to displayAsGF. it won't type check

guessRat(q: Symbol): GUESSER ==
-- opts := append(#2, [displayAsGF(false)$GuessOption, maxShift(0)$GuessOption, maxPower(1)$GuessOption, allDegrees(true)$GuessOption])
  opts := #2
guessHPaux(#1, (shiftHP q)(opts), opts)

guess(list: List F, guessers: List GUESSER, ops: List Symbol, options: LGOPT): GUESSRESULT ==
maxLevel := maxLevel(options)$GOPT0
xx := indexName(options)$GOPT0
if debug(options)$GOPT0 then
  output(hconcat("guessing level ", maxLevel::OutputForm))$OutputPackage
  output(hconcat("guessing ", list::OutputForm))$OutputPackage
res: GUESSRESULT := []
len := #list :: PositiveInteger
if len <= 1 then return res
for guesser in guessers repeat
  res := append(guesser(list, options), res)
  if debug(options)$GOPT0 then
    output(hconcat("res ", res::OutputForm))$OutputPackage
  if one(options)$GOPT0 and not empty? res then return res
  if (maxLevel = 0) then return res
if member?('guessProduct, ops) and not member?(0$F, list) then
  prodList: List F := [(list.(i+1))/(list.i) for i in 1..(len-1)]
-- tpd: maxLevel is NNI
  if not every?(one?, prodList) then
    var: Symbol := subscript('p, [len::OutputForm])
prodGuess :=
[[coerce(list.(guess.order+1))
  * product(guess.function, _
    equation(var, _
      (guess.order)::EXPR..xx::EXPR-1)), _
  guess.order] _
  for guess in guess(prodList, guessers, ops, options)$%]
-- tpd: this is broken
-- append([[indexName(var)$GuessOption)::Symbol,_
-- (maxLevel(maxLevel-1)$GuessOption)::NNI],_
-- options))$%]

if debug(options)$GOPT0 then
  output(hconcat("prodGuess "::OutputForm,
    prodGuess::OutputForm))
$OutputPackage

for guess in prodGuess
  | not any?(guess.function = #1.function, res) repeat
  res := cons(guess, res)

if one(options)$GOPT0 and not empty? res then return res

if member?('guessSum, ops) then
  sumList: List F := [(list.(i+1))-(list.i) for i in 1..(len-1)]
  -- tpd:maxLevel is NNI
  if not every?(#1=sumList.1, sumList) then
    var: Symbol := subscript('s, [len::OutputForm])
    sumGuess :=
      [[coerce(list.(guess.order+1)) _
        + summation(guess.function, _
          equation(var, _
            (guess.order)::EXPR..xx::EXPR-1)), _
        guess.order] _
        for guess in guess(sumList, guessers, ops, options)$%]
      -- tpd: this is broken
      -- for guess in guess(sumList, guessers, ops,_
      -- append([[indexName(var)$GuessOption)::Symbol,_
      -- (maxLevel(maxLevel-1)$GuessOption)::NNI],_
      -- options))$%]

  for guess in sumGuess
    | not any?(guess.function = #1.function, res) repeat
    res := cons(guess, res)

res

guess(list: List F): GUESSRESULT ==
guess(list, [guessADE$%, guessRec$%], ['guessProduct, 'guessSum], [])
package GuessAlgebraicNumber

— GuessAlgebraicNumber.input —

)set break resume
)sys rm -f GuessAlgebraicNumber.output
)spool GuessAlgebraicNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessAlgebraicNumber
--E 1

)spool
)lisp (bye)

— GuessAlgebraicNumber.help —

====================================================================
GuessAlgebraicNumber examples
====================================================================
This package exports guessing of sequences of rational functions

See Also:
- show GuessAlgebraicNumber

---

GuessAlgebraicNumber (GUESSAN)

Exports:
- diffHP
guess
guessADE
guessAlg
guessBinRat
guessExpRat
guessHP
guessHolo
guessPRec
guessPade
guessRat
guessRec
- shiftHP

---

)abbrev package GUESSAN GuessAlgebraicNumber
++ Description:
++ This package exports guessing of sequences of rational functions

GuessAlgebraicNumber() == Guess(AlgebraicNumber, AlgebraicNumber,
Expression Integer,
AlgebraicNumber,
id$MappingPackage1(AlgebraicNumber),
coerce$Expression(Integer))

---

"GUESSAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSAN"]
package GUESSF GuessFinite

— GuessFinite.input —

)set break resume
)sys rm -f GuessFinite.output
)spool GuessFinite.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessFinite
--E 1

)spool
)lisp (bye)

— GuessFinite.help —

====================================================================
GuessFinite examples
====================================================================

This package exports guessing of sequences of numbers in a finite field

See Also:
o )show GuessFinite

———
GuessFinite (GUESSF)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

package GUESSF GuessFinite

)abbrev package GUESSF GuessFinite
++ Description:
++ This package exports guessing of sequences of numbers in a finite field

GuessFinite(F:Join(FiniteFieldCategory, ConvertibleTo Integer)) ==
  Guess(F, F, Expression Integer, Integer, coerce$F,
  F2EXPRR$GuessFiniteFunctions(F))

package GUESSF1 GuessFiniteFunctions

)set break resume
GuessFiniteFunctions (GUESSF1)

Exports:
F2EXPRR
package GUESSF1 GuessFiniteFunctions

)abbrev package GUESSF1 GuessFiniteFunctions
++ Description:
++ This package exports guessing of sequences of numbers in a finite field

GuessFiniteFunctions(F:Join(FiniteFieldCategory, ConvertibleTo Integer)): Exports == Implementation where

EXPRR ==> Expression Integer

Exports == with

F2EXPRR: F -> EXPRR

Implementation == add

F2EXPRR(p: F): EXPRR == convert(p)@Integer::EXPRR

package GUESSINT GuessInteger

package GUESSINT GuessInteger
GuessInteger (GUESSINT)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

-- concerning algebraic functions, it may make sense to look at A.J. van der Poorten, Power Series Representing Algebraic Functions, Section 6.

)abbrev package GUESSINT GuessInteger
++ Description:
++ This package exports guessing of sequences of rational numbers
CHAPTER 8. CHAPTER G

GuessInteger() == Guess(Fraction Integer, Integer, Expression Integer, Fraction Integer, id$MappingPackage1(Fraction Integer), coerce$Expression(Integer))

---

— GUESSINT.dotabb —

"GUESSINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSINT"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSINT" -> "GUESS"

---

package GUESSP GuessPolynomial

— GuessPolynomial.input —

)set break resume
)sys rm -f GuessPolynomial.output
)spool GuessPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessPolynomial
--E 1

)spool
)lisp (bye)

---

— GuessPolynomial.help —

====================================================================
GuessPolynomial examples
====================================================================

This package exports guessing of sequences of rational functions
See Also:
o )show GuessPolynomial

---

GuessPolynomial (GUESSP)

Exports:
diffHP guess guessADE guessAlg guessBinRat
guessExpRat guessHP guessHolo guessPRec guessPade
guessRat guessRec shiftHP

— package GUESSP GuessPolynomial —

)abbrev package GUESSP GuessPolynomial
++ Description:
++ This package exports guessing of sequences of rational functions

GuessPolynomial() == Guess(Fraction Polynomial Integer, Polynomial Integer,
Expression Integer,
Fraction Polynomial Integer,
id$MappingPackage1(Fraction Polynomial Integer),
coerce$Expression(Integer))

——

— GUESSP.dotabb —

"GUESSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSP"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSP" -> "GUESS"
package GUESSUP GuessUnivariatePolynomial

---

--- GuessUnivariatePolynomial.input ---

)set break resume
)sys rm -f GuessUnivariatePolynomial.output
)spool GuessUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessUnivariatePolynomial
--E 1

)spool
)lisp (bye)

---

--- GuessUnivariatePolynomial.help ---

====================================================================
GuessUnivariatePolynomial examples
====================================================================

This package exports guessing of sequences of univariate rational functions

See Also:
o )show GuessUnivariatePolynomial

---
GuessUnivariatePolynomial (GUESSUP)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

--- package GUESSUP GuessUnivariatePolynomial ---

)abbrev package GUESSUP GuessUnivariatePolynomial
++ Description:
++ This package exports guessing of sequences of univariate rational functions

GuessUnivariatePolynomial(q: Symbol):Exports == Implementation where

| package GUESSUP GuessUnivariatePolynomial |

Exports == with

- guess: List F -> GUESSRESULT
  ++ \texttt{spad}(guess\ l) applies recursively \texttt{spadfun(guessRec)} and
  ++ \texttt{spadfun(guessADE)} to the successive differences and quotients of
  ++ the list. Default options as described in
  ++ \texttt{spadtype(GuessOptionFunctions0)} are used.

- guess: (List F, LGOPT) -> GUESSRESULT
  ++ \texttt{spad}(guess\ (l,\ options)) applies recursively \texttt{spadfun(guessRec)}
  ++ and \texttt{spadfun(guessADE)} to the successive differences and quotients
  ++ of the list. The given options are used.
guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ guessSum and quotients if ops contains the symbol guessProduct to
++ the list. Default options as described in
++ \spadtype{GuessOptionFunctions0} are used.

guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ \spad{guessSum} and quotients if ops contains the symbol
++ \spad{guessProduct} to the list. The given options are used.

guessExpRat: List F -> GUESSRESULT
++ \spad{guessExpRat l} tries to find a function of the form
++ n\rightarrow(\mathit{a}+\mathit{b} n)^n r(n), where r(n) is a rational function, that fits
++ l.

guessExpRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessExpRat(l, options)} tries to find a function of the
++ form n\rightarrow(\mathit{a}+\mathit{b} n)^n r(n), where r(n) is a rational function, that
++ fits l.

guessBinRat: List F -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form n\rightarrow\binomial(\mathit{a}+\mathit{b} n, n) r(n), where r(n) is a rational
++ function, that fits l.

guessBinRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form n\rightarrow\binomial(\mathit{a}+\mathit{b} n, n) r(n), where r(n) is a rational
++ function, that fits l.

-- if \ F has \ RetractableTo \ Symbol \ and \ S \ has \ RetractableTo \ Symbol \ then

guessExpRat: Symbol -> GUESSER
++ \spad{guessExpRat q} returns a guesser that tries to find a
++ function of the form n\rightarrow(\mathit{a}+\mathit{b} q^n)^n r(q^n), where r(q^n) is a
++ q-rational function, that fits l.

guessBinRat: Symbol -> GUESSER
++ \spad{guessBinRat q} returns a guesser that tries to find a
++ function of the form n\rightarrowq\binomial(\mathit{a}+\mathit{b} n, n) r(n), where r(q^n) is a
++ q-rational function, that fits l.

--------------------

guessHP: (LGOPT -> HPSPEC) -> GUESSER
++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
++ approximation to the series generated by the given function f.
guessADE: List F -> GUESSRESULT
  ++ \spad{guessADE 1} tries to find an algebraic differential equation
  ++ for a generating function whose first Taylor coefficients are
  ++ given by 1, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}.

guessADE: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessADE(l, options)} tries to find an algebraic
  ++ differential equation for a generating function whose first Taylor
  ++ coefficients are given by l, using the given options.

guessAlg: List F -> GUESSRESULT
  ++ \spad{guessAlg l} tries to find an algebraic equation for a
  ++ generating function whose first Taylor coefficients are given by
  ++ l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessADE}(l, maxDerivative == 0).

guessAlg: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessAlg(l, options)} tries to find an algebraic equation
  ++ for a generating function whose first Taylor coefficients are
  ++ given by l, using the given options. It is equivalent to
  ++ \spadfun{guessADE}(l, options) with \spad{maxDerivative == 0}.

guessHolo: List F -> GUESSRESULT
  ++ \spad{guessHolo l} tries to find an ordinary linear differential
  ++ equation for a generating function whose first Taylor coefficients
  ++ are given by l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessADE}(l, maxPower == 1).

guessHolo: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessHolo(l, options)} tries to find an ordinary linear differential
  ++ equation for a generating function whose first Taylor coefficients
  ++ are given by l, using the given options. It is equivalent to
  ++ \spadfun{guessADE}(l, options) with \spad{maxPower == 1}.

guessPade: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessPade(l, options)} tries to find a rational function
  ++ whose first Taylor coefficients are given by l, using the given
  ++ options. It is equivalent to \spadfun{guessADE}(l, maxDerivative == 0, maxPower == 1, allDegrees == true).

guessPade: List F -> GUESSRESULT
  ++ \spad{guessPade(l, options)} tries to find a rational function
  ++ whose first Taylor coefficients are given by l, using the default
  ++ options described in \spadtype{GuessOptionFunctions0}. It is
  ++ equivalent to \spadfun{guessADE}(l, options) with
guessRec: List F -> GUESSRESULT
++ \spad{guessRec 1} tries to find an ordinary difference equation whose first values are given by 1, using the default options described in \spadtype{GuessOptionFunctions0}.

guessRec: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessRec(1, options)} tries to find an ordinary difference equation whose first values are given by 1, using the given options.

guessPRec: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessPRec(1, options)} tries to find a linear recurrence with polynomial coefficients whose first values are given by 1, using the given options. It is equivalent to \spadfun{guessRec(1, options)} with \spad{maxPower == 1}.

guessPRec: List F -> GUESSRESULT
++ \spad{guessPRec 1} tries to find a linear recurrence with polynomial coefficients whose first values are given by 1, using the default options described in \spadtype{GuessOptionFunctions0}. It is equivalent to \spadfun{guessRec(1, maxPower == 1)}.

guessRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessRat(1, options)} tries to find a rational function whose first values are given by 1, using the given options. It is equivalent to \spadfun{guessRec(1, maxShift == 0, maxPower == 1, allDegrees == true)}.

guessRat: List F -> GUESSRESULT
++ \spad{guessRat 1} tries to find a rational function whose first values are given by 1, using the default options described in \spadtype{GuessOptionFunctions0}. It is equivalent to \spadfun{guessRec(1, maxShift == 0, maxPower == 1, allDegrees == true)}.

diffHP: LGOPT -> HPSPEC
++ \spad{diffHP options} returns a specification for Hermite-Pade approximation with the differential operator

shiftHP: LGOPT -> HPSPEC
++ \spad{shiftHP options} returns a specification for Hermite-Pade approximation with the shift operator

-- if F has RetractableTo Symbol and S has RetractableTo Symbol then

shiftHP: Symbol -> (LGOPT -> HPSPEC)
++ \spad{shiftHP options} returns a specification for
++ Hermite-Pade approximation with the $q$-shift operator

diffHP: Symbol -> (LGOPT -> HPSPEC)
++ \spad{diffHP options} returns a specification for Hermite-Pade
++ approximation with the $q$-dilation operator

guessRec: Symbol -> GUESSER
++ \spad{guessRec q} returns a guesser that finds an ordinary
++ q-difference equation whose first values are given by l, using
++ the given options.

guessPRec: Symbol -> GUESSER
++ \spad{guessPRec q} returns a guesser that tries to find
++ a linear q-recurrence with polynomial coefficients whose first
++ values are given by l, using the given options. It is
++ equivalent to \spadfun{guessRec}\spad{(q)} with
++ \spad{maxPower == 1}.

guessRat: Symbol -> GUESSER
++ \spad{guessRat q} returns a guesser that tries to find a
++ q-rational function whose first values are given by l, using
++ the given options. It is equivalent to \spadfun{guessRec} with
++ \spad{(l, maxShift == 0, maxPower == 1, allDegrees == true)}.

guessADE: Symbol -> GUESSER
++ \spad{guessADE q} returns a guesser that tries to find an
++ algebraic differential equation for a generating function whose
++ first Taylor coefficients are given by l, using the given
++ options.

Implementation == Guess(Fraction UP, UP,
   MyExpression(q, Integer),
   Fraction UP,
   id$MappingPackage1(Fraction UP),
   coerce$MyExpression(q, Integer))
Chapter 9

Chapter H

package HB HallBasis

— HallBasis.input —

)set break resume
)sys rm -f HallBasis.output
)spool HallBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show HallBasis
--E 1

)spool
)lisp (bye)

— HallBasis.help —

====================================================================
HallBasis examples
====================================================================

Generate a basis for the free Lie algebra on n generators over a ring R with identity up to basic commutators of length c using the algorithm of P. Hall as given in Serre's book Lie Groups -- Lie Algebras
See Also:
- \texttt{hallBasis}

---

HallBasis (HB)

Exports:
- \texttt{generate}
- \texttt{inHallBasis?}
- \texttt{lfunc}

--- package HB HallBasis ---

\texttt{\textbackslash abbrev package HB HallBasis}
\begin{verbatim}
++ Author : Larry Lambe
++ Date Created : August 1988
++ Date Last Updated : March 9 1990
++ Description:
++ Generate a basis for the free Lie algebra on \texttt{n}
++ generators over a ring \texttt{R} with identity up to basic commutators
++ of length \texttt{c} using the algorithm of P. Hall as given in Serre’s
++ book Lie Groups -- Lie Algebras

HallBasis() : Export == Implement where
  B ==> Boolean
  I ==> Integer
  NNI ==> NonNegativeInteger
  VI ==> Vector Integer
  VLI ==> Vector List Integer

Export ==> with
  lfunc : (I,I) -> I
  ++ lfunc(\texttt{d,n}) computes the rank of the \texttt{n}th factor in the
  ++ lower central series of the free \texttt{d}-generated free Lie
\end{verbatim}
++ algebra; This rank is d if n = 1 and binom(d,2) if
++ n = 2
inHallBasis? : (I,I,I,I) -> B
++ inHallBasis?(numberOfGens, leftCandidate, rightCandidate, left)
++ tests to see if a new element should be added to the P. Hall
++ basis being constructed.
++ The list \spad{[[leftCandidate, wt, rightCandidate]]}
++ is included in the basis if in the unique factorization of
++ rightCandidate, we have left factor leftOfRight, and
++ leftOfRight <= leftCandidate
generate : (NNI, NNI) -> VLI
++ generate(numberOfGens, maximalWeight) generates a vector of
++ elements of the form \spad{[left, weight, right]} which represents a
++ P. Hall basis element for the free lie algebra on numberOfGens
++ generators. We only generate those basis elements of weight
++ less than or equal to maximalWeight

Implement ==> add

lfunc(d,n) ==
n < 0 => 0
n = 0 => 1
n = 1 => d
sum:I := 0
m:I
for m in 1 .. (n-1) repeat
  if n rem m = 0 then
    sum := sum + m * lfunc(d,m)
res := (d**(n::NNI) - sum) quo n

inHallBasis?(n,i,j,l) ==
i => j => false
j <= n => true
l <= i => true
false

generate(n:NNI,c:NNI) ==
gens:=n
maxweight:=c
siz:I := 0
for i in 1 .. maxweight repeat siz := siz + lfunc(gens,i)
v:VLI := new(siz::NNI,[])
for i in 1 .. gens repeat v(i) := [0, 1, i]
firstindex:VI := new(maxweight::NNI,0)
wt:I := 1
firstindex(1) := 1
numComms:I := gens
newNumComms:I := numComms
done:B := false
while not done repeat
wt := wt + 1
if wt > maxweight then done := true
else
    firstindex(wt) := newNumComms + 1
    leftIndex := 1
    -- cW == complimentaryWeight
    cW:I := wt - 1
    while (leftIndex <= numComms) and (v(leftIndex).2 <= cW) repeat
        for rightIndex in firstindex(cW)...(firstindex(cW+1) - 1) repeat
            if inHallBasis?(gens,leftIndex,rightIndex,v(rightIndex).1) then
                newNumComms := newNumComms + 1
                v(newNumComms) := [leftIndex,wt,rightIndex]
            end if
        end for
        leftIndex := leftIndex + 1
        cW := wt - v(leftIndex).2
    end while
end if

---

HB.dotabb

"HB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=HB"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"HB" -> "IVECTOR"

---

package HEUGCD HeuGcd

--- HeuGcd.input ---

)set break resume
)sys rm -f HeuGcd.output
)spool HeuGcd.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show HeuGcd
--E 1

)spool
)lisp (bye)
HeuGcd examples

This package provides the functions for the heuristic integer gcd. Geddes's algorithm, for univariate polynomials with integer coefficients.

See Also:
o )show HeuGcd

HeuGcd (HEUGCD)

Exports:
  content  gcd  gcdcofact  gcdcofactprim  gcdprim  lintgcd

package HEUGCD HeuGcd

HeuGcd (BP):C == T
  where
  BP : UnivariatePolynomialCategory Integer
Z ==> Integer  
ContPrim ==> Record(cont:Z,prim:BP)

C == with
  gcd : List BP -> BP
  ++ gcd([f1,...,fk]) = gcd of the polynomials fi.
  ++
  ++X gcd([671*671*x^2-1,671*671*x^2+2*671*x+1])
  ++X gcd([7*x^2+1,(7*x^2+1)^2])

gcdprim : List BP -> BP
  ++ gcdprim([f1,...,fk]) = gcd of k PRIMITIVE univariate polynomials

gcdcofact : List BP -> List BP
  ++ gcdcofact([f1,...,fk]) = gcd and cofactors of k univariate polynomials.

gcdcofactprim: List BP -> List BP
  ++ gcdcofactprim([f1,...,fk]) = gcd and cofactors of k
  ++ primitive polynomials.

content : List BP -> List Z
  ++ content([f1,...,fk]) = content of a list of univariate polynomials

lintgcd : List Z -> Z
  ++ lintgcd([a1,...,ak]) = gcd of a list of integers

T == add

PI ==> PositiveInteger
NNI ==> NonNegativeInteger
Cases == Union("gcdprim","gcd","gcdcofactprim","gcdcofact")

import ModularDistinctDegreeFactorizer BP

-local functions
localgcd : List BP -> List BP
constNotZero : BP -> Boolean
height : BP -> PI
genpoly : (Z,PI) -> BP
negShiftz : (Z,PI) -> Z

internal : (Cases,List BP ) -> List BP
constcase : (List NNI ,List BP ) -> List BP
lincase : (List NNI ,List BP ) -> List BP

myNextPrime : ( Z , NNI ) -> Z

bigPrime:= prevPrime(2**26)$IntegerPrimesPackage(Integer)

myNextPrime(val:Z,bound:NNI) : Z == nextPrime(val)$IntegerPrimesPackage(Z)

constNotZero(f : BP ) : Boolean == (degree f = 0) and ~(zero? f)

negShiftz(n:Z,Modulus:PI):Z ==
  n < 0 => n:= n+Modulus
  n > (Modulus quo 2) => n-Modulus
--compute the height of a polynomial
height(f:BP):PI ==
k:PI:=1
while f^=0 repeat
  k:=max(k,abs(leadingCoefficient(f)@Z)::PI)
f:=reductum f
k

--reconstruct the polynomial from the value-adic representation of
--dval.
genpoly(dval:Z,value:PI):BP ==
d:=0$BP
val:=dval
for i in 0.. while (val<0) repeat
  val1:=negShiftz(val rem value,value)
  d:= d+monomial(val1,i)
  val:=(val-val1) quo value
  d

--gcd of a list of integers
lintgcd(lval:List(Z)):Z ==
  empty? lval => 0$Z
  member?(1,lval) => 1$Z
  lval:=sort((z1,z2) +-> z1<z2,lval)
  val:=lval.first
  for val1 in lval.rest while ^(val=1) repeat val:=gcd(val,val1)
  val

--content for a list of univariate polynomials
content(listf:List BP ):List(Z) ==
  [lintgcd coefficients f for f in listf]

--content of a list of polynomials with the relative primitive parts
contprim(listf:List BP ):List(ContPrim) ==
  [c:=lintgcd coefficients f,(f exquo c)::BP]$ContPrim for f in listf

-- one polynomial is constant, remark that they are primitive
-- but listf can contain the zero polynomial
constcase(listdeg:List NNI ,listf:List BP ): List BP ==
lind:=select(constNotZero,listf)
  empty? lind =>
    member?(1,listdeg) => lincase(listdeg,listf)
    localgcd listf
  or/[n>0 for n in listdeg] => cons(1$BP,listf)
lclistf:List(Z):= [leadingCoefficient f for f in listf]
d:=lintgcd(lclistf)
d=1 => cons(1$BP,listf)
cons(d::BP,[(lcf quo d)::BP for lcf in lclistf])
testDivide(listf: List BP, g:BP):Union(List BP, "failed") ==
result:=List BP := []
for f in listf repeat
    if (f1:=f exquo g) case "failed" then return "failed"
    result := cons(f1::BP,result)
reverse!(result)

-- one polynomial is linear, remark that they are primitive
lincase(listdeg:List NNI ,listf:List BP ):List BP ==
n:= position(1,listdeg)
g:=listf.n
result:=[g]
for f in listf repeat
    if (f1:=f exquo g) case "failed" then return cons(1$BP,listf)
    result := cons(f1::BP,result)
reverse(result)

IMG := InnerModularGcd(Z,BP,67108859,myNextPrime)

mindegpol(f:BP, g:BP):BP ==
degree(g) < degree (f) => g
f

-- local function for the gcd among n PRIMITIVE univariate polynomials
localgcd(listf:List BP ):List BP ==
hgt:"min"/[height(f) for f in listf|^zero? f]
answr:=2+2*hgt
minf := "mindegpol"/[f for f in listf|^zero? f]
(result := testDivide(listf, minf)) case List(BP) =>
    cons(minf, result::List BP)
if degree minf < 100 then for k in 1..10 repeat
    listval:=[f answr for f in listf]
dval:=lintgcd(listval)
dd:=genpoly(dval,answr)
contd:=content(dd)
d:=(dd exquo contd)::BP
result:List BP :=[d]
flag : Boolean := true
for f in listf while flag repeat
    (f1:=f exquo d) case "failed" => flag:=false
    result := cons (f1::BP,result)
if flag then return reverse(result)
nvalue:= answr*832040 quo 317811
if ((nvalue + answr) rem 2) = 0 then nvalue:=nvalue+1
answr:=nvalue::PI

gg:=modularGcdPrimitive(listf)$IMG
cons(gg,[(f exquo gg) :: BP for f in listf])

-- internal function: it evaluates the gcd and avoids duplication of
--code.
internal(flag:Cases,listf:List BP ):List BP ==
   --special cases
   listf=[] => [1$BP]
   (nlf:=#listf)=1 => [first listf,1$BP]
   minpol:=1$BP
   -- extract a monomial gcd
   mdeg:= "min"/[minimumDegree f for f in listf]
   if mdeg>0 then
      minpol1:= monomial(1,mdeg)
      listf:= [(f exquo minpol1)::BP for f in listf]
      minpol:=minpol*minpol1
   -- make the polynomials primitive
   Cgcd:List(Z):=[]
   contgcd : Z := 1
   if (flag case "gcd") or (flag case "gcdcofact") then
      contlistf:List(ContPrim):=contprim(listf)
      Cgcd:= [term.cont for term in contlistf]
      contgcd:=lintgcd(Cgcd)
      listf:List BP :=[term.prim for term in contlistf]
      minpol:=contgcd*minpol
   listdeg:= [degree f for f in listf ]
   f:= first listf
   if positiveRemainder(leadingCoefficient(f), bigPrime) ^= 0 then
      for g in rest listf repeat
         lcg := leadingCoefficient(g)
         if positiveRemainder(lcg, bigPrime) = 0 then
            leave
         f:=gcd(f,g,bigPrime)
      if degree f = 0 then return cons(minpol,listf)
   ans:List BP :=
      --one polynomial is constant
      member?(0,listdeg) => constcase(listdeg,listf)
      --one polynomial is linear
      member?(1,listdeg) => lincase(listdeg,listf)
      localgcd(listf)
   (result,ans):=(first ans*minpol,rest ans)
   if (flag case "gcdcofact") then
      ans:= [(p quo contgcd)*q for p in Cgcd for q in ans]
      cons(result,ans)
   --gcd among n PRIMITIVE univariate polynomials
   gcdprim (listf:List BP ):BP == first internal("gcdprim",listf)
   --gcd and cofactors for n PRIMITIVE univariate polynomials
   gcdcofactprim(listf:List BP ):List BP == internal("gcdcofactprim",listf)
   --gcd for n generic univariate polynomials.
   gcd(listf:List BP ): BP == first internal("gcd",listf)
gcd and cofactors for n generic univariate polynomials.
gcdcofact (listf:List BP):List BP == internal("gcdcofact",listf)
Chapter 10

Chapter I

package IDECOMP IdealDecompositionPackage

--- IdealDecompositionPackage.input ---

)set break resume
)sys rm -f IdealDecompositionPackage.output
)spool IdealDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IdealDecompositionPackage
--E 1

)spool
)lisp (bye)

--- IdealDecompositionPackage.help ---

====================================================================
IdealDecompositionPackage examples
====================================================================

This package provides functions for the primary decomposition of polynomial ideals over the rational numbers. The ideals are members of the PolynomialIdeals domain, and the polynomial generators are required to be from the DistributedMultivariatePolynomial domain.
See Also:
- )show IdealDecompositionPackage

---

IdealDecompositionPackage (IDECOMP)

Exports:
contract primaryDecomp prime? radical zeroDimPrimary? zeroDimPrime?

--- package IDECOMP IdealDecompositionPackage ---

)abbrev package IDECOMP IdealDecompositionPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Description:
++ This package provides functions for the primary decomposition of
++ polynomial ideals over the rational numbers. The ideals are members
++ of the \spadtype{PolynomialIdeals} domain, and the polynomial generators are
++ required to be from the \spadtype{DistributedMultivariatePolynomial} domain.

IdealDecompositionPackage(vl,nv) : C == T -- take away nv, now doesn't
-- compile if it isn't there

where
vl : List Symbol
nv : NonNegativeInteger
Z ==> Integer -- substitute with PFE cat
Q ==> Fraction Z
F ==> Fraction P
P ==> Polynomial Z
UP ==> SparseUnivariatePolynomial P
Expon ==> DirectProduct(nv,NNI)
PACKAGE IDECOMP IDEALDECOMPOSITIONPACKAGE

OV ==> OrderedVariableList(vl)
SE ==> Symbol
SUP ==> SparseUnivariatePolynomial(DPoly)

DPoly1 ==> DistributedMultivariatePolynomial(vl,Q)
DPoly ==> DistributedMultivariatePolynomial(vl,F)
NNI ==> NonNegativeInteger

Ideal == PolynomialIdeals(Q,Expon,OV,DPoly1)
FIdeal == PolynomialIdeals(F,Expon,OV,DPoly)
Fun0 == Union("zeroPrimDecomp","zeroRadComp")
GenPos == Record(changeval:List Z,genideal:FIdeal)

C == with

zeroDimPrime? : Ideal -> Boolean
++ zeroDimPrime?(I) tests if the ideal I is a 0-dimensional prime.

zeroDimPrimary? : Ideal -> Boolean
++ zeroDimPrimary?(I) tests if the ideal I is 0-dimensional primary.

prime? : Ideal -> Boolean
++ prime?(I) tests if the ideal I is prime.

radical : Ideal -> Ideal
++ radical(I) returns the radical of the ideal I.

classicalDecomp : Ideal -> List(Ideal)
++ classicalDecomp(I) returns a list of primary ideals such that their
++ intersection is the ideal I.

contract : (Ideal,List OV ) -> Ideal
++ contract(I,lvar) contracts the ideal I to the polynomial ring
++ \spad{F[lvar]}.

T == add

import MPolyCatRationalFunctionFactorizer(Expon,OV,Z,DPoly)
import GroebnerPackage(F,Expon,OV,DPoly)
import GroebnerPackage(Q,Expon,OV,DPoly1)

---- Local Functions -----
genPosLastVar : (FIdeal,List OV ) -> GenPos
zeroPrimDecomp : (FIdeal,List OV ) -> List(FIdeal)
zeroRadDecomp : (FIdeal,List OV ) -> FIdeal
zerodimcase : (FIdeal,List OV ) -> Boolean
is0dimprimary : (FIdeal,List OV ) -> Boolean
backGenPos : (FIdeal,List Z,List OV ) -> FIdeal
reduceDim : (Fun0,FIdeal,List OV ) -> List FIdeal
findvar : (FIdeal,List OV ) -> OV
testPower : (SUP,OV,FIdeal) -> Boolean
goodPower : (DPoly,FIdeal) -> Record(spol:DPoly,id:FIdeal)
CHAPTER 10. CHAPTER I

pushdown : (DPoly,OV) -> DPoly
pushdterm : (DPoly,OV,Z) -> DPoly
pushup : (DPoly,OV) -> DPoly
pushuterm : (DPoly,SE,OV) -> DPoly
pushucoef : (UP,OV) -> DPoly
trueden : (P,SE) -> P
rearrange : (List OV) -> List OV
deleteunit : List FIdeal -> List FIdeal
ismonic : (DPoly,OV) -> Boolean

MPCFQF ==> MPolyCatFunctions2(OV,Expon,Expon,Q,F,DPoly1,DPoly)
MPCFFQ ==> MPolyCatFunctions2(OV,Expon,Expon,F,Q,DPoly,DPoly1)

convertQF(a:Q) : F == ((numer a)::F)/((denom a)::F)
convertFQ(a:F) : Q == (ground numer a)/(ground denom a)

internalForm(I:Ideal) : FIdeal ==
  Id:=generators I
  nId:=\[map(convertQF,poly)\$MPCFQF for poly in Id\]
  if groebner? I then groebnerIdeal nId
  ideal nId

externalForm(I:FIdeal) : Ideal ==
  Id:=generators I
  nId:=\[map(convertFQ,poly)\$MPCFFQ for poly in Id\]
  if groebner? I then groebnerIdeal nId
  ideal nId

lvint:=[variable(xx)::OV for xx in vl]
nvint1:=(#lvint-1)::NNI

deleteunit(lI: List FIdeal) : List FIdeal ==
  [I for I in lI | _^ element?(1$DPoly,I)]

rearrange(vlist:List OV) : List OV ==
  vlist[] => vlist
  sort((z1,z2)+->z1>z2,setDifference(lvint,setDifference(lvint,vlist)))

--- radical of a 0-dimensional ideal ----
zeroRadComp(I:FIdeal,truelist:List OV) : FIdeal ==
  truelist=[] => I
  Id:=generators I
  x:OV:=truelist.last
  if #Id=1 then
    f:=Id.first
    g:= (f exquo (gcd (f,differentiate(f,x)))::DPoly
         groebnerIdeal([g]))
    y:=truelist.first
    px:DPoly:=x::DPoly
   ```
py:DPoly:=y::DPoly
f:=Id.last
g:= (f exquo (gcd (f,differentiate(f,x))))::DPoly
Id:=groebner(cons(g,remove(f,Id)))
lf:=Id.first
pv:DPoly:=0
pw:DPoly:=0
while degree(lf,y)\^=1 repeat
  val:=random()$\mathbb{Z}$ rem 23
  pv:=px+val*py
  pw:=px-val*py
  Id:=groebner([(univariate(h,x)).pv for h in Id])
  lf:=Id.first
ris:= generators(zeroRadComp(groebnerIdeal(Id.rest),truelist.rest))
ris:=cons(lf,r)is
if pv\^=0 then
  ris:=[(univariate(h,x)).pw for h in ris]
groebnerIdeal(groebner ris)

----- find the power that stabilizes (I:s) -----
goodPower(s:DPoly,I:FIdeal) : Record(spol:DPoly,id:FIdeal) ==
f:DPoly:=s
I:=groebner I
J:=generators(JJ:= (saturate(I,s)))
while \^ in?(ideal([f*g for g in J]),I) repeat f:=s*f
[f,JJ]

----- is the ideal zerodimensional? -----
zerodimcase(J:FIdeal,truelist:List OV) : Boolean ==
element?(1,J) => true
truelist=[] => true
n:=#truelist
Jd:=groebner generators J
for x in truelist while Jd=[] repeat
  f := Jd.first
  Jd:=Jd.rest
  if \((y:=mainVariable f) case "failed" or (y::OV \^=x )
    or \^ (ismonic (f,x)) then return false
  while Jd=[] and (mainVariable Jd.first)::OV=x repeat Jd:=Jd.rest
  if Jd=[] and position(x,truelist)<n then return false
  true

----- choose the variable for the reduction step -----  
findvar(J:FIdeal,truelist:List OV) : OV ==
lmonicvar:List OV :=[]
for f in generators J repeat
  t:=f - reductum f
  vt:List OV :=variables t

if #vt=1 then lmonicvar:=setUnion(vt,lmonicvar)
badvar:=setDifference(truelist,lmonicvar)
badvar.first

---- function for the "reduction step" ----
reduceDim(flag:Fun0,J:FIdeal,truelist:List OV) : List(FIdeal) ==
element?(1,J) => [J]
zerodimcase(J,truelist) =>
  (flag case "zeroPrimDecomp") => zeroPrimDecomp(J,truelist)
  (flag case "zeroRadComp") => [zeroRadComp(J,truelist)]
x:OV:=findvar(J,truelist)
Jnew:=[pushdown(f,x) for f in generators J]
JC: List Fideal :=[]
JC:=reduceDim(flag,groebnerIdeal Jnew,remove(x,truelist))
res1:=[ideal([pushup(f,x) for f in generators idp]) for idp in JC]
s:=pushup((_.*/[leadingCoefficient f for f in Jnew]):DPoly,x)
degree(s,x)=0 => res1
res1:=[saturate(II,s) for II in res1]
good:=goodPower(s,J)
sideal := groebnerIdeal(groebner(cons(good.spol,generators J)))
in?(good.id, sideal) => res1
sresult:=reduceDim(flag,sideal,truelist)
for JJ in sresult repeat
  if not(in?(good.id,JJ)) then res1:=cons(JJ,res1)
res1

---- Primary Decomposition for 0-dimensional ideals ----
zeroPrimDecomp(I:FIdeal,truelist:List OV): List(FIdeal) ==
truelist=[] => list I
newJ:=genPosLastVar(I,truelist);lval:=newJ.changeval;
J:=groebner newJ.genideal
x:=truelist.last
Jd:=generators J
g:=Jd.last
lifact:= factors factor(g)
ris:List Fideal:=[]
for ef in lifact repeat
  g:=DPoly:=(ef.factor)**(ef.exponent::NNI)
  J1:= groebnerIdeal(groebner cons(g,Jd))
  if _"(is0dimprimary (J1,truelist)) then
    return zeroPrimDecomp(I,truelist)
  ris:=cons(groebner backGenPos(J1,lval,truelist),ris)
ris

---- radical of an Ideal ----
radiical(I:Ideal) : Ideal ==
J:=groebner(internalForm I)
truelist:=rearrange("setUnion"/[variables f for f in generators J])
truelist=[] => externalForm J
externalForm("intersect"/reduceDim("zeroRadComp",J,truelist))
-- the following functions are used to "push" x in the coefficient ring --

----- push x in the coefficient domain for a polynomial -----
pushdown(g:DPoly,x:OV) : DPoly ==
rf:DPoly:=0$DPoly
i:=position(x,lvint)
while g^=0 repeat
  g1:=reductum g
  rf:=rf+pushdterm(g-g1,x,i)
  g := g1
rf

----- push x in the coefficient domain for a term -----
pushdterm(t:DPoly,x:OV,i:Z):DPoly ==
n:=degree(t,x)
xp:=convert(x)@SE
cf:=monomial(1,xp,n)$P :: F
newt := t exquo monomial(1,x,n)$DPoly
cf * newt::DPoly

----- push back the variable -----
pushup(f:DPoly,x:OV) :DPoly ==
h:=1$P
rf:DPoly:=0$DPoly
g := f
xp := convert(x)@SE
while g^=0 repeat
  h:=lcm(trueden(denom leadingCoefficient g,xp),h)
  g:=reductum g
  f:=(h::F)*f
while f^=0 repeat
  g:=reductum f
  rf:=rf+pushuterm(f-g,xp,x)
  f:=g
rf

trueden(c:P,x:SE) : P ==
degree(c,x) = 0 => 1
  c

----- push x back from the coefficient domain for a term -----
pushuterm(t:DPoly,xp:SE,x:OV):DPoly ==
pushucoef((univariate(numer leadingCoefficient t,xp)$P), x)*
  monomial(inv((denom leadingCoefficient t)::F),degree t)$DPoly

pushucoef(c:UP,x:OV):DPoly ==
c = 0 => 0
monomial((leadingCoefficient c)::F::DPoly,x,degree c) + pushucoef(reductum c,x)

-- is the 0-dimensional ideal I primary ? --
----- internal function ----
is0dimprimary(J:FIdeal,truelist:List OV) : Boolean ==
  element?(1,J) => true
  Jd:=generators(groebner J)
  #(factors factor Jd.last)**=1 => return false
  i:=subtractIfCan(#truelist,1)
  (i case "failed") => return true
  JR:=(reverse Jd);JM:=groebnerIdeal([JR.first]);JP:List(DPoly):=[]
  for f in JR.rest repeat
    if _^ ismonic(f,truelist.i) then
      if _^ inRadical?(f,JM) then return false
      JP:=cons(f,JP)
    else
      x:=truelist.i
      i:=(i-1)::NNI
      if _^ testPower(univariate(f,x),x,JM) then return false
      JM :=groebnerIdeal(append(cons(f,JP),generators JM))
  true

---- Functions for the General Position step ----

---- put the ideal in general position ----
genPosLastVar(J:FIdeal,truelist:List OV):GenPos ==
  x := last truelist ;lv1:List OV :=remove(x,truelist)
  ranvals:List(Z):=[(random()$Z rem 23) for vv in lv1]
  val:=+/
    [rv*(vv::DPoly) for vv in lv1 for rv in ranvals]
  val:=val+(x::DPoly)
  [ranvals,groebnerIdeal(groebner([univariate(p,x)].val
    for p in generators J))]]$GenPos

---- convert back the ideal ----
backGenPos(I:FIdeal,lval:List Z,truelist:List OV) : FIdeal ==
  lval=[] => I
  x := last truelist ;lv1:List OV:=remove(x,truelist)
  val:=-(_+/
    [rv*(vv::DPoly) for vv in lv1 for rv in lval])
  val:=val+(x::DPoly)
  groebnerIdeal
    (groebner([univariate(p,x)].val for p in generators J]))

ismonic(f:DPoly,x:OV) : Boolean ==
  ground? leadingCoefficient(univariate(f,x))

---- test if f is power of a linear mod (rad J) ----
----- f is monic ----- 
testPower(uf:SUP,x:OV,J:FIdeal) : Boolean ==
df:=degree(uf)
trailp:DPoly := inv(df:Z ::F)*coefficient(uf,(df-1)::NNI)
linp:SUP:=(monomial(1$DPoly,1$NNI)$SUP +
    monomial(trailp,0$NNI)$SUP)**df
g:DPoly:=multivariate(uf-linp,x)
inRadical?(g,J)

----- Exported Functions -----

-- is the 0-dimensional ideal I prime ? --
zeroDimPrime?(I:Ideal) : Boolean ==
J:=groebner((genPosLastVar(internalForm I,lvint)).genideal)
element?(1,J) => true
n:NNI:=#v1;i:NNI:=1
Jd:=generators J
#Jd^=n => false
for f in Jd repeat
    if _ ismonic(f,lvint.i) then return false
    if i<n and (degree univariate(f,lvint.i))^=1 then return false
    i:=i+1
g:=Jd.n
#(lfact:=factors(factor g)) >1 => false
lfact.1.exponent =1

-- is the 0-dimensional ideal I primary ? --
zeroDimPrimary?(J:Ideal):Boolean ==
is0dimprimary(internalForm J,lvint)

------ Primary Decomposition of I ------

primaryDecomp(I:Ideal) : List(Ideal) ==
J:=groebner(internalForm I)
truelist:=rearrange("setUnion"/[variables f for f in generators J])
truelist=[] => [externalForm J]
[externalForm II for II in reduceDim("zeroPrimDecomp",J,truelist)]

----- contract I to the ring with lvar variables ----

contract(I:Ideal,lvar: List OV) : Ideal ==
Id:= generators(groebner I)
empty?(Id) => I
fullVars:="setUnion"/[variables g for g in Id]
fullVars = lvar => I
n:= # lvar
#fullVars < n => error "wrong vars"
n=0 => I
newVars:=
    append([vv for vv in fullVars| ~member?(vv,lvar)]$List(OV),lvar)
subsVars := [monomial(1,vv,1)$DPoly1 for vv in newVars]
1J:= [eval(g,fullVars,subsVars) for g in Id]
J := groebner(1J)
J=[1] => groebnerIdeal J
J=[0] => groebnerIdeal empty()
J:=[f for f in J| member?(mainVariable(f)::OV,newVars)]
fullPol :=[monomial(1,vv,1)$DPoly1 for vv in fullVars]
groebnerIdeal([eval(gg,newVars,fullPol) for gg in J])

package INCRMAPS IncrementingMaps

--- IncrementingMaps.input ---

)set break resume
)sys rm -f IncrementingMaps.output
)spool IncrementingMaps.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show IncrementingMaps
--E 1

)spool
)lisp (bye)

--- IncrementingMaps.help ---

====================================================================
IncrementingMaps examples
====================================================================
This package provides operations to create incrementing functions.
See Also:
o )show IncrementingMaps

---

IncrementingMaps (INCRMAPS)

INCRMAPS

SGROUP

Exports:
    increment  incrementBy

--- package INCRMAPS IncrementingMaps ---

)abbrev package INCRMAPS IncrementingMaps
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations to create incrementing functions.

IncrementingMaps(R:Join(Monoid, AbelianSemiGroup)): with
    increment: () -> (R -> R)
    ++ increment() produces a function which adds \spad{1} to whatever
    ++ argument it is given. For example, if \{f := increment()\} then
    ++ \spad{f x} is \spad{x+1}.
    incrementBy: R -> (R -> R)
    ++ incrementBy(n) produces a function which adds \spad{n} to whatever
    ++ argument it is given. For example, if \{f := increment(n)\} then
    ++ \spad{f x} is \spad{x+n}.

== add
    increment()  == x +-> 1 + x
incrementBy \( n \Rightarrow x \mapsto n + x \)

---

--- 

package INFPRODO InfiniteProductCharacteristicZero

--- InfiniteProductCharacteristicZero.input ---

)set break resume
)sys rm -f InfiniteProductCharacteristicZero.output
)spool InfiniteProductCharacteristicZero.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteProductCharacteristicZero
--E 1

)spool
)lisp (bye)

---

--- InfiniteProductCharacteristicZero.help ---

=================================================================
InfiniteProductCharacteristicZero examples
=================================================================

This package computes infinite products of univariate Taylor series over an integral domain of characteristic 0.

See Also:
  o )show InfiniteProductCharacteristicZero
InfiniteProductCharacteristicZero (INFPROD0)

Exports:
- evenInfiniteProduct
- infiniteProduct
- oddInfiniteProduct
- generalInfiniteProduct

--- package INFPROD0 InfiniteProductCharacteristicZero ---

```spad
)abbrev package INFPROD0 InfiniteProductCharacteristicZero
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over an integral domain of characteristic 0.

InfiniteProductCharacteristicZero(Coef,UTS):~
Exports == Implementation where
  Coef : Join(IntegralDomain,CharacteristicZero)
  UTS : UnivariateTaylorSeriesCategory Coef
  I ==> Integer

Exports ==> with

  infiniteProduct: UTS -> UTS
  ++ infiniteProduct(f(x)) computes \(\prod_{n=1,2,3\ldots} f(x^n)\).
  ++ The series \(f(x)\) should have constant coefficient 1.
  evenInfiniteProduct: UTS -> UTS
  ++ evenInfiniteProduct(f(x)) computes \(\prod_{n=2,4,6\ldots} f(x^n)\).
  ++ The series \(f(x)\) should have constant coefficient 1.
  oddInfiniteProduct: UTS -> UTS
  ++ oddInfiniteProduct(f(x)) computes \(\prod_{n=1,3,5\ldots} f(x^n)\).
  ++ The series \(f(x)\) should have constant coefficient 1.
```
generalInfiniteProduct: (UTS,I,I) -> UTS
++ generalInfiniteProduct(f(x),a,d) computes
++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
++ The series \spad{f(x)} should have constant coefficient 1.

Implementation ==> add

import StreamInfiniteProduct Coef

infiniteProduct x == series infiniteProduct coefficients x
evenInfiniteProduct x == series evenInfiniteProduct coefficients x
oddInfiniteProduct x == series oddInfiniteProduct coefficients x

generalInfiniteProduct(x,a,d) ==
    series generalInfiniteProduct(coefficients x,a,d)

—— INFPRODO.dotabb ——

"INFPRODO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFPRODO"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"INFPRODO" -> "UTSCAT"

————

package INPRODFFF InfiniteProductFiniteField

—— InfiniteProductFiniteField.input ——

)set break resume
)sys rm -f InfiniteProductFiniteField.output
)spool InfiniteProductFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteProductFiniteField
--E 1

)spool
)lisp (bye)
This package computes infinite products of univariate Taylor series over an arbitrary finite field.

See Also:
- )show InfiniteProductFiniteField

Exports:
- evenInfiniteProduct
- generalInfiniteProduct
- infiniteProduct
- oddInfiniteProduct

InfiniteProductFiniteField(K,UP,Coef,UTS):=
Exports == Implementation where
K : Join(Field,Finite,ConvertibleTo Integer)
UP : UnivariatePolynomialCategory K
Coef : MonogenicAlgebra(K,UP)
UTS : UnivariateTaylorSeriesCategory Coef
I ==> Integer
RN ==> Fraction Integer
SAE ==> SimpleAlgebraicExtension
ST ==> Stream
STF ==> StreamTranscendentalFunctions
STT ==> StreamTaylorSeriesOperations
ST2 ==> StreamFunctions2
SUP ==> SparseUnivariatePolynomial

Exports ==> with

infiniteProduct: UTS -> UTS
++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
++ The series \spad{f(x)} should have constant coefficient 1.
evenInfiniteProduct: UTS -> UTS
++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
++ The series \spad{f(x)} should have constant coefficient 1.
oddInfiniteProduct: UTS -> UTS
++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
++ The series \spad{f(x)} should have constant coefficient 1.
generalInfiniteProduct: (UTS,I,I) -> UTS
++ generalInfiniteProduct(f(x),a,d) computes
++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
++ The series \spad{f(x)} should have constant coefficient 1.

Implementation ==> add

liftPoly: UP -> SUP RN
liftPoly poly ==
  -- lift coefficients of 'poly' to integers
  ans : SUP RN := 0
  while not zero? poly repeat
    coef := convert(leadingCoefficient poly)@I :: RN
    ans := ans + monomial(coef,degree poly)
    poly := reductum poly
  ans

reducePoly: SUP RN -> UP
reducePoly poly ==
  -- reduce coefficients of 'poly' to elements of K
  ans : UP := 0
  while not zero? poly repeat
    coef := numer(leadingCoefficient(poly)) :: K
    ans := ans + monomial(coef,degree poly)
    poly := reductum poly
  ans
POLY := liftPoly definingPolynomial()$Coef
ALG := SAE(RN,SUP RN,POLY)

infiniteProduct x ==
  stUP := map(lift,coefficients x)$ST2(Coef,UP)
  stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
  stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
  stALG := exp(lambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
  stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
  stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
  series map(reduce,stUP)$ST2(UP,Coef)

evenInfiniteProduct x ==
  stUP := map(lift,coefficients x)$ST2(Coef,UP)
  stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
  stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
  stALG := exp(evenlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
  stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
  stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
  series map(reduce,stUP)$ST2(UP,Coef)

oddInfiniteProduct x ==
  stUP := map(lift,coefficients x)$ST2(Coef,UP)
  stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
  stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
  stALG := exp(oddlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
  stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
  stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
  series map(reduce,stUP)$ST2(UP,Coef)

generalInfiniteProduct(x,a,d) ==
  stUP := map(lift,coefficients x)$ST2(Coef,UP)
  stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
  stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
  stALG := generalLambert(log(stALG)$STF(ALG),a,d)$STT(ALG)
  stALG := exp(stALG)$STF(ALG)
  stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
  stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
  series map(reduce,stUP)$ST2(UP,Coef)
package INPRODPF InfiniteProductPrimeField

-- InfiniteProductPrimeField.input --

)set break resume
)sys rm -f InfiniteProductPrimeField.output
)spool InfiniteProductPrimeField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteProductPrimeField
--E 1

)spool
)lisp (bye)

-- InfiniteProductPrimeField.help --

====================================================================
InfiniteProductPrimeField examples
====================================================================

This package computes infinite products of univariate Taylor series over a field of prime order.

See Also:
  o )show InfiniteProductPrimeField
InfiniteProductPrimeField (INPRODPF)

Exports:

evenInfiniteProduct generalInfiniteProduct infiniteProduct oddInfiniteProduct

class InfiniteProductPrimeField (INPRODPF)

++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over a field of prime order.

InfiniteProductPrimeField(Coef,UTS): Exports == Implementation where
Coef : Join(Field,Finite,ConvertibleTo Integer)
UTS : UnivariateTaylorSeriesCategory Coef
I ==> Integer
ST ==> Stream

Exports ==> with

infiniteProduct: UTS -> UTS
++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.  
++ The series \spad{f(x)} should have constant coefficient 1.
evenInfiniteProduct: UTS -> UTS
++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.  
++ The series \spad{f(x)} should have constant coefficient 1.
oddInfiniteProduct: UTS -> UTS
++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.  
++ The series \spad{f(x)} should have constant coefficient 1.
generalInfiniteProduct: (UTS,I,I) -> UTS
++ generalInfiniteProduct(f(x),a,d) computes
++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.  
++ The series \spad{f(x)} should have constant coefficient 1.
Implementation ==> add

```lisp
import StreamInfiniteProduct Integer

applyOverZ( (ST I -> ST I, ST Coef) -> ST Coef
applyOverZ(f, st) ==
  stZ := map(z1 +-> convert(z1)@Integer, st)$StreamFunctions2(Coef, I)
  map(z1 +-> z1 :: Coef, f stZ)$StreamFunctions2(I, Coef)

infiniteProduct x ==
  series applyOverZ(infiniteProduct, coefficients x)

evenInfiniteProduct x ==
  series applyOverZ(evenInfiniteProduct, coefficients x)

oddInfiniteProduct x ==
  series applyOverZ(oddInfiniteProduct, coefficients x)

generalInfiniteProduct(x, a, d) ==
  series
    applyOverZ(
      (z1: ST(I)): ST(I) +-> generalInfiniteProduct(z1, a, d), coefficients x)
```

———

INPRODPF.dotabb ——

"INPRODPF" [color="#FF4488", href="bookvol10.4.pdf#nameddest=INPRODPF"]
"UTSCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=UTSCAT"]
"INPRODPF" -> "UTSCAT"

———

package ITFUN2 InfiniteTupleFunctions2

—— InfiniteTupleFunctions2.input ——

```lisp
)set break resume
)sys rm -f InfiniteTupleFunctions2.output
)spool InfiniteTupleFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteTupleFunctions2
--E 1
```
InfiniteTupleFunctions2 (ITFUN2)

Exports:
map

--- package ITFUN2 InfiniteTupleFunctions2 ---

)abbrev package ITFUN2 InfiniteTupleFunctions2
++ Description:
++ Functions defined on streams with entries in two sets.

InfiniteTupleFunctions2(A:Type,B:Type): Exports == Implementation where
  IT  => InfiniteTuple
Exports ==> with
map: ((A -> B), IT A) -> IT B
++ \spad{map(f,[x0,x1,x2,...])} returns \spad{[f(x0),f(x1),f(x2),...]}.

Implementation ==> add

map(f,x) ==
map(f,x pretend Stream(A))$StreamFunctions2(A,B) pretend IT(B)

— ITFUN2.dotabb —

"ITFUN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITFUN2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITFUN2" -> "TYPE"

package ITFUN3 InfiniteTupleFunctions3

— InfiniteTupleFunctions3.input —

)set break resume
)sys rm -f InfiniteTupleFunctions3.output
)spool InfiniteTupleFunctions3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteTupleFunctions3
--E 1

)spool
)lisp (bye)

— InfiniteTupleFunctions3.help —

====================================================================
InfiniteTupleFunctions3 examples
====================================================================
Functions defined on streams with entries in two sets.
See Also:
o )show InfiniteTupleFunctions3

InfiniteTupleFunctions3 (ITFUN3)

Exports:
map

--- package ITFUN3 InfiniteTupleFunctions3 ---

)abbrev package ITFUN3 InfiniteTupleFunctions3
++ Description:
++ Functions defined on streams with entries in two sets.
InfiniteTupleFunctions3(A:Type, B:Type,C:Type): Exports == Implementation where
IT  ==> InfiniteTuple
ST  ==> Stream
SF3  ==> StreamFunctions3(A,B,C)
FUN  ==> ((A,B)->C)
Exports ==> with
  map: (((A,B)->C), IT A, IT B) -> IT C
  ++ map(f,a,b) undocumented
  map: (((A,B)->C), ST A, IT B) -> ST C
  ++ map(f,a,b) undocumented
  map: (((A,B)->C), IT A, ST B) -> ST C

++ map(f,a,b) \ undocumented  

Implementation ==> add

map(f:FUN, s1:IT A, s2:IT B):IT C ==
  map(f, s1 pretend Stream(A), s2 pretend Stream(B))$SF3 pretend IT(C)

map(f:FUN, s1:ST A, s2:IT B):ST C ==
  map(f, s1, s2 pretend Stream(B))$SF3

map(f:FUN, s1:IT A, s2:ST B):ST C ==
  map(f, s1 pretend Stream(A), s2)$SF3

---

package INFINITY Infinity

--- Infinity.input ---

)set break resume
)sys rm -f Infinity.output
)spool Infinity.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Infinity
--E 1

)spool
)lisp (bye)

---

--- Infinity.help ---

====================================================================
Infinity examples
====================================================================
Default infinity signatures for the interpreter;

See Also:
o )show Infinity

Infinity (INFINITY)

Exports:
infinity minusInfinity plusInfinity

— package INFINITY Infinity —

)abbrev package INFINITY Infinity
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Top-level infinity
++ Default infinity signatures for the interpreter;

Infinity(): with
  infinity : () -> OnePointCompletion Integer
    + infinity() returns infinity.
  plusInfinity : () -> OrderedCompletion Integer
    + plusInfinity() returns plusInfinity.
  minusInfinity: () -> OrderedCompletion Integer
    + minusInfinity() returns minusInfinity.
== add
infinity() == infinity()$OnePointCompletion(Integer)
plusInfinity() == plusInfinity()$OrderedCompletion(Integer)
minusInfinity() == minusInfinity()$OrderedCompletion(Integer)

---

--- INFINITY.dotabb ---

"INFINITY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFINITY"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"INFINITY" -> "PID"
"INFINITY" -> "OAGROUP"

---

package IALGFACT InnerAlgFactor

--- InnerAlgFactor.input ---

)set break resume
)sys rm -f InnerAlgFactor.output
)spool InnerAlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerAlgFactor
--E 1

)spool
)lisp (bye)

---

--- InnerAlgFactor.help ---

====================================================================
InnerAlgFactor examples
====================================================================

Factorisation in a simple algebraic extension. Factorization of
univariate polynomials with coefficients in an algebraic extension of a field over which we can factor UP's;

See Also:
o )show InnerAlgFactor

InnerAlgFactor (IALGFACT)

Exports:
factor

— package IALGFACT InnerAlgFactor —

)abbrev package IALGFACT InnerAlgFactor
++ Author: Patrizia Gianni
++ Date Last Updated: 20 Jul 1988
++ Description:
++ Factorisation in a simple algebraic extension
++ Factorization of univariate polynomials with coefficients in an algebraic extension of a field over which we can factor UP's;

InnerAlgFactor(F, UP, AlExt, AlPol): Exports == Implementation where
  F : Field
  UP : UnivariatePolynomialCategory F
  AlPol: UnivariatePolynomialCategory AlExt
  AlExt : Join(Field, CharacteristicZero, MonogenicAlgebra(F,UP))
  NUP ==> SparseUnivariatePolynomial UP
  N ==> NonNegativeInteger
  Z ==> Integer
  FR ==> Factored UP
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2
Exports ==> with
factor: (AlPol, UP -> FR) -> Factored AlPol
  ++ factor(p, f) returns a prime factorisation of p;
  ++ f is a factorisation map for elements of UP;

Implementation ==> add
pnorm : AlPol -> UP
convt : AlPol -> NUP
change : UP -> AlPol
perturbfactor: (AlPol, Z, UP -> FR) -> List AlPol
irrfactor : (AlPol, Z, UP -> FR) -> List AlPol

perturbfactor(f, k, fact) ==
  pol := monomial(1$AlExt,1)-
    monomial(reduce monomial(k::F,1)$UP,0)
  newf := elt(f, pol)
  lsols := irrfactor(newf, k, fact)
  pol := monomial(1, 1) +
    monomial(reduce monomial(k::F,1)$UP,0)
  [elt(pp, pol) for pp in lsols]

--- factorize the square-free parts of f ---
irrfactor(f, k, fact) ==
  degree(f) =$N 1 => [f]
  newf := f
  nn := pnorm f
  --newval:RN:=1
  --pert:=false
  --if ~ SqFr? nn then
    -- pert:=true
    -- newterm:=perturb(f)
    -- newf:=newterm.ppol
    -- newval:=newterm.pval
    -- nn:=newterm.nnorm
  listfact := factors fact nn
  #listfact =$N 1 =>
    first(listfact).exponent =$Z 1 => [f]
  perturbfactor(f, k + 1, fact)
  listerm:List(AlPol):= []
  for pelt in listfact repeat
    g := gcd(change(pelt.factor), newf)
    newf := (newf exquo g)::AlPol
    listerm :=
      pelt.exponent =$Z 1 => cons(g, listerm)
      append(perturbfactor(g, k + 1, fact), listerm)
  listerm
factor(f, fact) ==
  sqf := squareFree f
  unit(sqf) * _/*/[_/*/[primeFactor(pol, sqterm.exponent)
    for pol in irrFactor(sqterm.factor, 0, fact)]
    for sqterm in factors sqf]

p := definingPolynomial()$AlExt
newp := map(x -> x::UP, p)$UPCF2(F, UP, UP, NUP)

pnorm q == resultant(convrt q, newp)
change q == map(coerce, q)$UPCF2(F,UP,AlExt,AlPol)

convrt q ==
  swap(map(lift, q)$UPCF2(AlExt, AlPol,
    UP, NUP))$CommuteUnivariatePolynomialCategory(F, UP, NUP)

---

package ICDEN InnerCommonDenominator

---
InnerCommonDenominator (ICDEN)

Exports:
clearDenominator  commonDenominator  splitDenominator

-- package ICDEN InnerCommonDenominator --

)abbrev package ICDEN InnerCommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Description:
++ InnerCommonDenominator provides functions to compute
++ the common denominator of a finite linear aggregate of elements
++ of the quotient field of an integral domain.
InnerCommonDenominator(R, Q, A, B): Exports == Implementation where
   R: IntegralDomain
   Q: QuotientFieldCategory R
   A: FiniteLinearAggregate R
   B: FiniteLinearAggregate Q

Exports ==> with
   commonDenominator: B -> R
      ++ commonDenominator([q1,...,qn]) returns a common denominator
      ++ d for q1,...,qn.
   clearDenominator : B -> A
      ++ clearDenominator([q1,...,qn]) returns \spad{[p1,...,pn]} such that
      ++ \spad{qi = pi/d} where d is a common denominator for the qi’s.
   splitDenominator : B -> Record(num: A, den: R)
      ++ splitDenominator([q1,...,qn]) returns
      ++ \spad{[[p1,...,pn], d]} such that
      ++ \spad{qi = pi/d} and d is a common denominator for the qi’s.

Implementation ==> add
   import FiniteLinearAggregateFunctions2(Q, B, R, A)

   clearDenominator l ==
      d := commonDenominator l
      map(x +-> numer(d*x), l)

   splitDenominator l ==
      d := commonDenominator l
      [map(x +-> numer(d*x), l), d]

if R has GcdDomain then
   commonDenominator l == reduce(lcm, map(denom, l), 1)
else
   commonDenominator l == reduce("*", map(denom, l), 1)

--- ICDEN.dotabb ---
"ICDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ICDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ICDEN" -> "PFECAT"

---
package IMATLIN InnerMatrixLinearAlgebraFunctions

— InnerMatrixLinearAlgebraFunctions.input —

)set break resume
)sys rm -f InnerMatrixLinearAlgebraFunctions.output
)spool InnerMatrixLinearAlgebraFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerMatrixLinearAlgebraFunctions
--E 1

)spool
)lisp (bye)

— InnerMatrixLinearAlgebraFunctions.help —

====================================================================
InnerMatrixLinearAlgebraFunctions examples
====================================================================

InnerMatrixLinearAlgebraFunctions is an internal package which provides standard linear algebra functions on domains in MatrixCategory.

See Also:
  o )show InnerMatrixLinearAlgebraFunctions

———
InnerMatrixLinearAlgebraFunctions (IMATLIN)

Exports:

--- package IMATLIN InnerMatrixLinearAlgebraFunctions ---

)abbrev package IMATLIN InnerMatrixLinearAlgebraFunctions  
++ Author: Clifton J. Williamson, P.Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: September 1993
++ Description:
++ \spadtype{InnerMatrixLinearAlgebraFunctions} is an internal package
++ which provides standard linear algebra functions on domains in
++ \spad{MatrixCategory}

InnerMatrixLinearAlgebraFunctions(R,Row,Col,M):=
    Exports == Implementation where
    R : Field
    Row : FiniteLinearAggregate R
    Col : FiniteLinearAggregate R
    M : MatrixCategory(R,Row,Col)
    I == Integer

Exports ==> with
    rowEchelon: M -> M
    +++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.
    rank: M -> NonNegativeInteger
    +++ \spad{rank(m)} returns the rank of the matrix m.
    nullity: M -> NonNegativeInteger
++ \spad{nullity(m)} returns the nullity of the matrix m. This is the ++ dimension of the null space of the matrix m.
if Col has shallowlyMutable then
  nullSpace: M -> List Col
    ++ \spad{nullSpace(m)} returns a basis for the null space of the ++ matrix m.

determinant: M -> R
  ++ \spad{determinant(m)} returns the determinant of the matrix m. +++ an error message is returned if the matrix is not square.

generalizedInverse: M -> M
  ++ \spad{generalizedInverse(m)} returns the generalized (Moore--Penrose) ++ inverse of the matrix m, i.e. the matrix h such that ++ m*h*m = h, h*m*h = m, m*h and h*m are both symmetric matrices.

inverse: M -> Union(M,"failed")
  ++ \spad{inverse(m)} returns the inverse of the matrix m. ++ If the matrix is not invertible, "failed" is returned. ++ Error: if the matrix is not square.

Implementation ==> add

rowAllZeroes?: (M,I) -> Boolean
rowAllZeroes?(x,i) ==
  -- determines if the ith row of x consists only of zeroes
  -- internal function: no check on index i
  for j in minColIndex(x)..maxColIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

colAllZeroes?: (M,I) -> Boolean
colAllZeroes?(x,j) ==
  -- determines if the ith column of x consists only of zeroes
  -- internal function: no check on index j
  for i in minRowIndex(x)..maxRowIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

rowEchelon y ==
  -- row echelon form via Gaussian elimination
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  i := minR
  n: I := minR - 1
  for j in minC..maxC repeat
    if i > maxR => return x
    n := minR - 1
    -- n = smallest k such that k >= i and x(k,j) ^= 0
    for k in i..maxR repeat
      if qelt(x,k,j) ^= 0 then leave (n := k)
    n = minR - 1 => "no non-zeroes"
-- put nth row in ith position
if i ^= n then swapRows_!(x,i,n)
-- divide ith row by its first non-zero entry
b := inv qelt(x,i,j)
qsetelt_!(x,i,j,1)
for k in (j+1)..maxC repeat qsetelt_!(x,i,k,b * qelt(x,i,k))
-- perform row operations so that jth column has only one 1
for k in minR..maxR repeat
  if k ^= i and qelt(x,k,j) ^= 0 then
    for k1 in (j+1)..maxC repeat
      qsetelt_!(x,k,k1,qelt(x,k,k1) - qelt(x,k,j) * qelt(x,i,k1))
      qsetelt_!(x,k,j,0)
  -- increment i
  i := i + 1
x

rank x ==
y :=
  (rk := nrows x) > (rh := ncols x) =>
  rk := rh
  transpose x
  copy x
  y := rowEchelon y; i := maxRowIndex y
  while rk > 0 and rowAllZeroes?(y,i) repeat
    i := i - 1
    rk := (rk - 1) :: NonNegativeInteger
  rk :: NonNegativeInteger
nullity x == (ncols x - rank x) :: NonNegativeInteger

if Col has shallowlyMutable then

nullSpace y ==
x := rowEchelon y
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
nrow := nrows x; ncol := ncols x
basis : List Col := nil()
rk := nrow; row := maxR
-- compute rank = # rows - # rows of all zeroes
while rk > 0 and rowAllZeroes?(x,row) repeat
  rk := (rk - 1) :: NonNegativeInteger
  row := (row - 1) :: NonNegativeInteger
-- if maximal rank, return zero vector
  ncol <= nrow and rk = ncol => [new(ncol,0)]
-- if rank = 0, return standard basis vectors
  rk = 0 =>
    for j in minC..maxC repeat
      w : Col := new(ncol,0)
      qsetelt_!(w,j,1)
basis := cons(w,basis)
basis
-- v contains information about initial 1's in the rows of x
-- if the ith row has an initial 1 in the jth column, then
-- v.j = i; v.j = minR - 1, otherwise
v : IndexedOneDimensionalArray(I,minC) := new(ncol,minR - 1)
for i in minR..(minR + rk - 1) repeat
  for j in minC.. while qelt(x,i,j) = 0 repeat j
    qsetelt_!(v,j,i)
  j := maxC; l := minR + ncol - 1
while j >= minC repeat
  w : Col := new(ncol,0)
  -- if there is no row with an initial 1 in the jth column,
  -- create a basis vector with a 1 in the jth row
  if qelt(v,j) = minR - 1 then
    colAllZeroes?(x,j) =>
      qsetelt_!(w,l,1)
basis := cons(w,basis)
  for k in minC..(j-1) for ll in minR..(l-1) repeat
    if qelt(v,k) ^= minR - 1 then
      qsetelt_!(w,ll,-qelt(x,qelt(v,k),j))
basis := cons(w,basis)
  j := j - 1; l := l - 1

basis

determinant y ==
  (ndim := nrows y) ^= (ncols y) =>
    error "determinant: matrix must be square"
-- Gaussian Elimination
ndim = 1 => qelt(y,minRowIndex y,minColIndex y)
x := copy y
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
ans : R := 1
for i in minR..(maxR - 1) for j in minC..(maxC - 1) repeat
  if qelt(x,i,j) = 0 then
    rown := minR - 1
    for k in (i+1)..maxR repeat
      qelt(x,k,j) ^= 0 => leave (rown := k)
    if rown = minR - 1 then return 0
    swapRows_!(x,i,rown); ans := -ans
  ans := qelt(x,i,j) * ans; b := -inv qelt(x,i,j)
  for l in (j+1)..maxC repeat
    qsetelt_!(x,i,l,b * qelt(x,i,l))
  for k in (i+1)..maxR repeat
    if (b := qelt(x,k,j)) ^= 0 then
      for l in (j+1)..maxC repeat
        qsetelt_!(x,k,l,qelt(x,k,l) + b * qelt(x,i,l))
  qelt(x,maxR,maxC) * ans
generalizedInverse(x) ==
SUP:=SparseUnivariatePolynomial R
FSUP := Fraction SUP
VFSUP := Vector FSUP
MATCAT2 := MatrixCategoryFunctions2(R, Row, Col, M, FSUP, VFSUP, VFSUP, Matrix FSUP)
MATCAT22 := MatrixCategoryFunctions2(FSUP, VFSUP, VFSUP, Matrix FSUP, R, Row, Col, M)
y:= map((r1:R):FSUP +-> coerce(coerce(r1)$SUP)$(Fraction SUP),x)$MATCAT2
ty:=transpose y
yy:=ty*y
nc:=ncols yy
var:=monomial(1,1)$SUP ::(Fraction SUP)
yy:=inverse(yy+scalarMatrix(ncols yy,var))::Matrix(FSUP)*ty
map((z1:FSUP):R +-> elt(z1,0),yy)$MATCAT22

inverse x ==
(ndim := nrows x) ^= (ncols x) =>
   error "inverse: matrix must be square"
ndim = 2 =>
   ans2 : M := zero(ndim, ndim)
   zero?(det := x(1,1)*x(2,2)-x(1,2)*x(2,1)) => "failed"
   detinv := inv det
   ans2(1,1) := x(2,2)*detinv
   ans2(1,2) := -x(1,2)*detinv
   ans2(2,1) := -x(2,1)*detinv
   ans2(2,2) := x(1,1)*detinv
   ans2
AB : M := zero(ndim,ndim + ndim)
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
kmin := minRowIndex AB; kmax := kmin + ndim - 1
lmin := minColIndex AB; lmax := lmin + ndim - 1
for i in minR..maxR for k in kmin..kmax repeat
   for j in minC..maxC for l in lmin..lmax repeat
      qsetelt_!(AB,k,l,qelt(x,i,j))
      qsetelt_!(AB,k,lmin + ndim + k - kmin,1)
   AB := rowEchelon AB
elt(AB,kmax,lmax) = 0 => "failed"
subMatrix(AB,kmin,kmax,lmin + ndim,lmax + ndim)
package IMATQF InnerMatrixQuotientFieldFunctions

InnerMatrixQuotientFieldFunctions examples

InnerMatrixQuotientFieldFunctions provides functions on matrices over an integral domain which involve the quotient field of that integral domain. The functions rowEchelon and inverse return matrices with entries in the quotient field.

See Also:
- )show InnerMatrixQuotientFieldFunctions
INNERMATRIXQUOTIENTFIELDFUNCTIONS

InnerMatrixQuotientFieldFunctions (IMATQF)

Exports:
inverse rowEchelon nullSpace

--- package IMATQF InnerMatrixQuotientFieldFunctions ---

)abbrev package IMATQF InnerMatrixQuotientFieldFunctions
++ Author: Clifton J. Williamson
++ Date Created: 22 November 1989
++ Date Last Updated: 22 November 1989
++ Description:
++ \spadtype{InnerMatrixQuotientFieldFunctions} provides functions on matrices
++ over an integral domain which involve the quotient field of that integral
++ domain. The functions rowEchelon and inverse return matrices with
++ entries in the quotient field.

InnerMatrixQuotientFieldFunctions(R,Row,Col,M,QF,Row2,Col2,M2):_
Exports == Implementation where
R : IntegralDomain
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)
QF : QuotientFieldCategory R
Row2 : FiniteLinearAggregate QF
Col2 : FiniteLinearAggregate QF
M2 : MatrixCategory(QF,Row2,Col2)
IMATLIN ==> InnerMatrixLinearAlgebraFunctions(QF,Row2,Col2,M2)
MATCAT2 ==> MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Col2,M2)
CDEN ==> InnerCommonDenominator(R,QF,Col,Col2)

Exports ==> with
rowEchelon: M -> M2
++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.
++ the result will have entries in the quotient field.
inverse: M -> Union(M2,"failed")
++ \spad{inverse(m)} returns the inverse of the matrix m.
++ If the matrix is not invertible, "failed" is returned.
++ Error: if the matrix is not square.
++ Note that the result will have entries in the quotient field.
if Col2 has shallowlyMutable then
nullSpace : M -> List Col
  ++ \spad{nullSpace(m)} returns a basis for the null space of the
  ++ matrix m.
Implementation ==> add

qfMat: M -> M2
qfMat m == map((r1:R):QF +-> r1::QF,m)$MATCAT2
rowEchelon m == rowEchelon(qfMat m)$IMATLIN
inverse m ==
  (inv := inverse(qfMat m)$IMATLIN) case "failed" => "failed"
  inv :: M2
if Col2 has shallowlyMutable then
nullSpace m ==
  [clearDenominator(v)$CDEN for v in nullSpace(qfMat m)$IMATLIN]

— IMATQF.dotabb —
"IMATQF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IMATQF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"IMATQF" -> "PFECAT"

package INMODGCD InnerModularGcd

— InnerModularGcd.input —

)set break resume
)sys rm -f InnerModularGcd.output
)spool InnerModularGcd.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerModularGcd
This file contains the functions for modular gcd algorithm for
univariate polynomials with coefficients in a non-trivial euclidean
domain (i.e. not a field). The package parametrised by the
coefficient domain, the polynomial domain, a prime, and a function for
choosing the next prime

See Also:
o )show InnerModularGcd

Exports:
modularGcd modularGcdPrimitive reduction
++ Author: J.H. Davenport and P. Gianni
++ Date Created: July 1990
++ Date Last Updated: November 1991
++ Description:
++ This file contains the functions for modular gcd algorithm
++ for univariate polynomials with coefficients in a
++ non-trivial euclidean domain (i.e. not a field).
++ The package parametrised by the coefficient domain,
++ the polynomial domain, a prime, and a function for choosing the next prime

InnerModularGcd(R,BP,pMod,nextMod):C == T
where
  Z ==> Integer
  NNI ==> NonNegativeInteger
  R : EuclideanDomain
  BP : UnivariatePolynomialCategory(R)
  pMod : R
  nextMod : (R,NNI) -> R

C == with
  modularGcdPrimitive : List BP -> BP
    ++ modularGcdPrimitive(f1,f2) computes the gcd of the two polynomials
    ++ f1 and f2 by modular methods.
  modularGcd : List BP -> BP
    ++ modularGcd(listf) computes the gcd of the list of polynomials
    ++ listf by modular methods.
  reduction : (BP,R) -> BP
    ++ reduction(f,p) reduces the coefficients of the polynomial f
    ++ modulo the prime p.

T == add

    -- local functions --
    height : BP -> NNI
    mbound : (BP,BP) -> NNI
    modGcdPrimitive : (BP,BP) -> BP
    test : (BP,BP,BP) -> Boolean
    merge : (R,R) -> Union(R,"failed")
    modInverse : (R,R) -> R
    exactquo : (BP,BP,R) -> Union(BP,"failed")
    constNotZero : BP -> Boolean
    constcase : (List NNI ,List BP ) -> BP
    lincase : (List NNI ,List BP ) -> BP

if R has IntegerNumberSystem then
  reduction(u:BP,p:R):BP ==
    p = 0 => u
    map((r1:R):R +-> symmetricRemainder(r1,p),u)
else
    reduction(u:BP,p:R):BP ==
    p = 0 => u
    map((r1:R):R +-> r1 rem p,u)

FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)
zeroChar : Boolean := R has CharacteristicZero

-- exported functions --

-- modular Gcd for a list of primitive polynomials
modularGcdPrimitive(listf : List BP) :BP ==
    empty? listf => 0$BP
    g := first listf
    for f in rest listf | `zero? f while degree g > 0 repeat
        g:=modGcdPrimitive(g,f)
    g

-- gcd for univariate polynomials
modularGcd(listf : List BP): BP ==
    listf:=remove!(0$BP,listf)
    empty? listf => 0$BP
    # listf = 1 => first listf
    minpol:=1$BP
    -- extract a monomial gcd
    mdeg:= "min"/[minimumDegree f for f in listf]
    if mdeg>0 then
        minpol1:= monomial(1,mdeg)
        listf:= [(f exquo minpol1)::BP for f in listf]
        minpol:=minpol*minpol1
    listdeg:=[degree f for f in listf ]
    -- make the polynomials primitive
    listCont := [(content f for f in listf)]
    contgcd:= gcd listCont
    -- make the polynomials primitive
    listf :=[(f exquo cf)::BP for f in listf for cf in listCont]
    minpol:=contgcd*minpol
    ans:BP :=
        --one polynomial is constant
        member?(1,listf) => 1
        --one polynomial is linear
        member?(1,listdeg) => lincase(listdeg,listf)
        modularGcdPrimitive listf
    minpol*ans

-- local functions --

--one polynomial is linear, remark that they are primitive
lincase(listdeg:List NNI ,listf:List BP ) : BP ==
    n:= position(1,listdeg)
g := listf.n
for f in listf repeat
    if (f1 := f exquo g) case "failed" then return 1$BP
    g

-- test if d is the gcd
test(f:BP, g:BP, d:BP): Boolean ==
    d0 := coefficient(d, 0)
    coefficient(f, 0) exquo d0 case "failed" => false
    coefficient(g, 0) exquo d0 case "failed" => false
    f exquo d case "failed" => false
    g exquo d case "failed" => false
    true

-- gcd and cofactors for PRIMITIVE univariate polynomials
-- also assumes that constant terms are non zero
modGcdPrimitive(f:BP, g:BP): BP ==
    df := degree f
    dg := degree g
    dp := FP
    lcf := leadingCoefficient f
    lcg := leadingCoefficient g
    testdeg := NNI
    lcd := gcd(lcf, lcg)
    prime := pMod
    bound := mbound(f, g)
    while zero? (lcd rem prime) repeat
        prime := nextMod(prime, bound)
        soFar := gcd(reduce(f, prime), reduce(g, prime))::BP
        testdeg := degree soFar
        zero? testdeg => return 1$BP
        ldp := FP :=
            (lcp := leadingCoefficient(soFar::BP)) = 1) =>
                reduce(lcp::BP, prime)
                reduce((modInverse(lcp, prime) (*) lcd)::BP, prime)
        soFar := reduce(ldp::BP * soFar, prime)::BP
        soFarModulus := prime
        -- choose the prime
        while true repeat
            prime := nextMod(prime, bound)
            lcd rem prime = 0 => "next prime"
            fp := reduce(f, prime)
            gp := reduce(g, prime)
            dp := gcd(fp, gp)
            dgp := euclideanSize dp
            if dgp = 0 then return 1$BP
            if dgp = dg and ^(f exquo g case "failed") then return g
            if dgp = df and ^(g exquo f case "failed") then return f
            dgp > testdeg => "next prime"
            ldp := FP :=
```lisp
PACKAGE INMODGCD INNERMODULARGCD

((lcp::leadingCoefficient(dp::BP)) = 1) =>
  reduce(lcd::BP,prime)
reduce((modInverse(lcd,prime) * lcd)::BP,prime)
dp:=1 * dp
dgp=testdeg =>
correction:=reduce(dp::BP-soFar,prime)::BP
zero? correction =>
  ans:=reduce(lcd::BP*soFar,soFarModulus)::BP
  cont:=content ans
  ans:=(ans exquo cont)::BP
  test(f,g,ans) => return ans
  soFarModulus:=soFarModulus*prime
correctionFactor:=modInverse(soFarModulus rem prime,prime)
  -- the initial rem is just for efficiency
  soFar:=soFar*soFarModulus*(correctionFactor*correction)
  soFarModulus:=soFarModulus*prime
  soFar:=reduce(soFar,soFarModulus)::BP

dgp<testdeg =>
  soFarModulus:=prime
  soFar:=dp::BP
  testdeg:=dgp
  if not zeroChar and euclideanSize(prime)>1 then
    result:=dp::BP
    test(f,g,result) => return result
    -- this is based on the assumption that the caller of this package,
    -- in non-zero characteristic, will use primes of the form
    -- x-alpha as long as possible, but, if these are exhausted,
    -- will switch to a prime of degree larger than the answer
    -- so the result can be used directly.

merge(p:R,q:R):Union(R,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

modInverse(c:R,p:R):R ==
  (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

  invlc::modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlc*v,p))
  reduction(r.remainder,p) ^=0 => "failed"
  reduction(invlc*r.quotient,p)

-- compute the height of a polynomial
height(f:BP):NNI ==
  defg:=degree f
  "max"/[euclideanSize cc for cc in coefficients f]
```
-- compute the bound
mbound(f:BP,g:BP):NNI ==
hf:=height f
hg:=height g
2*min(hf,hg)

---

%----------------------------------------------------------
\section{package FOMOGCD ForModularGcd}
\-- ForModularGcd(R,BP) : C == T
\-- where
\-- R : EuclideanDomain -- characteristic 0
\-- BP : UnivariatePolynomialCategory(R)
\--
\-- C == with
\-- nextMod : (R,NNI) -> R
\--
\-- T == add
\-- nextMod(val:R,bound:NNI) : R ==
\-- ival:Z:= val pretend Z
\-- (nextPrime(ival)$IntegerPrimesPackage(Z))::R
\--
\-- ForTwoGcd(F) : C == T
\-- where
\-- F : Join(Finite,Field)
\-- SUP ==> SparseUnivariatePolynomial
\-- R ==> SUP F
\-- P ==> SUP R
\-- UPCF2 ==> UnivariatePolynomialCategoryFunctions2
\--
\-- C == with
\-- nextMod : (R,NNI) -> R
\--
\-- T == add
\-- nextMod(val:R,bound:NNI) : R ==
\-- ris:R:= nextItem(val) :: R
\-- euclideanSize ris < 2 => ris
\-- generateIrredPoly(
\-- (bound+1)::PositiveInteger)$IrredPolyOverFiniteField(F)
\--
\--
\-- ModularGcd(R,BP) == T
\-- where
\-- R : EuclideanDomain -- characteristic 0
\-- BP : UnivariatePolynomialCategory(R)
\-- T ==> InnerModularGcd(R,BP,67108859::R,nextMod$ForModularGcd(R,BP))
PACKAGE INNMFACT INNERMULTFACT

-- TwoGcd(F) : C == T
-- where
-- F  : Join(Finite,Field)
-- SUP ==> SparseUnivariatePolynomial
-- R  ==> SUP F
-- P  ==> SUP R
--
-- T => InnerModularGcd(R,P,nextMod(monomial(1,1)$R)$ForTwoGcd(F),
--          nextMod$ForTwoGcd(F))

INMODGCD.dotabb

"INMODGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INMODGCD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INMODGCD" -> "PFECAT"

package INNMFACT InnerMultFact

InnerMultFact.input

)set break resume
)sys rm -f InnerMultFact.output
)spool InnerMultFact.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerMultFact
--E 1

)spool
)lisp (bye)

InnerMultFact.help

InnerMultFact examples

This is an inner package for factoring multivariate polynomials over various coefficient domains in characteristic 0. The univariate factor operation is passed as a parameter. Multivariate hensel lifting is used to lift the univariate factorization

See Also:
- )show InnerMultFact

---

**InnerMultFact (INNMFACT)**

--- package INNMFACT InnerMultFact ---

)abbrev package INNMFACT InnerMultFact
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is an inner package for factoring multivariate polynomials
++ over various coefficient domains in characteristic 0.
++ The univariate factor operation is passed as a parameter.
++ Multivariate hensel lifting is used to lift the univariate
++ factorization

-- Both exposed functions call mFactor. This deals with issues such as
-- monomial factors, contents, square-freeness etc., then calls intfact.
-- This uses intChoose to find a "good" evaluation and factorise the
-- corresponding univariate, and then uses MultivariateLifting to find
-- the multivariate factors.

InnerMultFact(OV,E,R,P) : C == T
where
  R : Join(EuclideanDomain, CharacteristicZero)
    -- with factor on R[x]
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)
  BP ==> SparseUnivariatePolynomial R
  UFactor ==> (BP -> Factored BP)
  Z ==> Integer
  MParFact ==> Record(irr:P,pow:Z)
  USP ==> SparseUnivariatePolynomial P
  SUParFact ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact ==> Record(contp:R,factors:List MParFact)

-- contp = content,
-- factors = List of irreducible factors with exponent
L ==> List
C == with
  factor : (P,UFactor) -> Factored P
  ++ factor(p,ufact) factors the multivariate polynomial p
  ++ by specializing variables and calling the univariate
  ++ factorizer ufact.
  factor : (USP,UFactor) -> Factored USP
  ++ factor(p,ufact) factors the multivariate polynomial p
  ++ by specializing variables and calling the univariate
  ++ factorizer ufact. p is represented as a univariate
  ++ polynomial with multivariate coefficients.
T == add
  NNI ==> NonNegativeInteger
  LeadFact ==> Record(polfac:L P,correct:R,corrfact:L BP)
  ContPrim ==> Record(cont:P,prim:P)
  ParFact ==> Record(irr:BP,pow:Z)
  FinalFact ==> Record(contp:R,factors:L ParFact)
  NewOrd ==> Record(npol:USP,nvar:L OV,newdeg:L NNI)
  pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R
import GenExEuclid(R,BP)
import MultivariateLifting(E,OV,R,P)
import FactoringUtilities(E,OV,R,P)
import LeadingCoefDetermination(OV,E,R,P)
Valuf ==> Record(inval:L L R,unvfact:L BP,lu:R,complead:L R)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2
---- Local Functions ----

mFactor : (P,UFactor) -> MFinalFact
supFactor : (USP,UFactor) -> SUPFinalFact
mfconst : (USP,LOV,LNNI,UFactor) -> LUSP
mfpol : (USP,LOV,LNNI,UFactor) -> LUSP
monicMfpol: (USP,LOV,LNNI,UFactor) -> LUSP
varChoose : (P,LOV,LNNI) -> NewOrd
simplify : (P,LOV,LNNI,UFactor) -> MFinalFact
intChoose : (USP,LOV,R,LP,LLR,UFactor) -> Union(Valuf,"failed")
intfact : (USP,LOV,LNNI,MFinalFact,LLR,UFactor) -> LUSP
pretest : (P,NNI,LOV,LR) -> FinalFact
checkzero : (USP,BP) -> Boolean
localNorm : LBP -> Z

convertPUP(lfg:MFinalFact): SUPFinalFact ==
[lfg.contp,[[lff.irr ::USP,lff.pow]$SUParFact
  for lff in lfg.factors]]$SUPFinalFact

-- intermediate routine if an SUP was passed in.
supFactor(um:USP,ufactor:UFactor) : SUPFinalFact ==
ground?(um) => convertPUP(mFactor(ground um,ufactor))
lvar:LOV := "setUnion"/[variables cf for cf in coefficients um]
empty? lvar => -- the polynomial is univariate
  umv:= map(ground,um)$UPCF2(P,USP,R,BP)
  lfact:=ufactor umv
  [retract unit lfact,[[(map(coerce,ff.factor)$UPCF2(R,BP,P,USP),
    ff.exponent] for ff in factors lfact]]$SUPFinalFact
lcont:P
flead : SUPFinalFact:=[0,empty()]]$SUPFinalFact
factorlist:L SUParFact :=empty()

mdeg :=minimumDegree um ---- is the Mindeg > 0? ----
if mdeg>0 then
  f1:USP:=monomial(1,mdeg)
  um:=(um exquo f1)::USP
  factorlist:=cons([monomial(1,1),mdeg],factorlist)
  if degree um=0 then return
  lfg:=convertPUP mFactor(ground um, ufactor)
  [lfg.contp,append(factorlist,lfg.factors)]
uum:=unitNormal um
um :=uum.canonical
sqfacs := squareFree(um)$MultivariateSquareFree(E,OV,R,P)
lcont := ground(uum.unit * unit sqfacs)
  ---- Factorize the content ----
flead:=convertPUP mFactor(lcont,ufactor)
factorlist:=append(flead.factors,factorlist)
  ---- Make the polynomial square-free ----
sqqfact:=factors sqfacs
for fact in sqqfact repeat
  ffactor:=fact.factor
  ffexp:=fact.exponent
  zero? degree ffactor =>
    lfg:=mFactor(ground ffactor,ufactor)
    lcont:=lfg.contp * lcont
    factorlist := append(factorlist,
        [[lff.irr ::USP,lff.pow * ffexp]$SUParFact
          for lff in lfg.factors])
  coefs := coefficients ffactor
  ldeg:= "max"/[(degree(fc,xx) for fc in coefs) for xx in lvar]
  lf :=
    ground?(leadingCoefficient ffactor) =>
      mfconst(ffactor,lvar,ldeg,ufactor)
      mfpol(ffactor,lvar,ldeg,ufactor)
    auxfl:=[lfp,ffexp]$SUParFact for lfp in lf]
    factorlist:=append(factorlist,auxfl)
  lcfacs := */[leadingCoefficient leadingCoefficient(f irr)**((f.pow)::NNI)
    for f in factorlist]
  
  [(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R,
    factorlist]$SUPFinalFact

factor(um:USP,ufactor:UFactor):Factored USP ==
  flist := supFactor(um,ufactor)
  (flist.contp):: P :: USP *
    (*/[primeFactor(u.irr,u.pow) for u in flist.factors])

checkzero(u:USP,um:BP) : Boolean ==
  u=0 => um =0
  um = 0 => false
  degree u = degree um => checkzero(reductum u, reductum um)
  false

  --- Choose the variable of less degree ---
varChoose(m:P,lvar:L OV,ldeg:L NNI) : NewOrd ==
  k:="min"/[d for d in ldeg]
  k=degree(m,first lvar) =>
    [univariate(m,first lvar),lvar,ldeg]$NewOrd
  i:=position(k,ldeg)
  x:OV:=lvar.i
  ldeg:=cons(k,delete(ldeg,i))
  lvar:=cons(x,delete(lvar,i))
  [univariate(m,x),lvar,ldeg]$NewOrd

localNorm(lum: L BP): Z ==
  R is AlgebraicNumber =>
    "max"/[numberOfMonomials ff for ff in lum]
    "max"/[+/[euclideanSize cc for i in 0..degree ff |
      (cc:= coefficient(ff,i))'=0] for ff in lum]
--- Choose the integer to reduce to univariate case ---
intChoose(um:USP,lvar:LOV,clc:R,plist:L P,ltry:LLR,
ufactor:UFactor) : Union(Valuf, "failed") ==

-- declarations
degum:NNI := degree um
nvar1:=#lvar
range:NNI:=5
unifact:L BP
ctf1 : R := 1
testp:Boolean := -- polynomial leading coefficient
    empty? plist => false
    true
leadcomp,leadcomp1 : L R
leadcomp:=leadcomp1:=empty()
nfatt:NNI := degum+1
lffc:R:=1
lffc1:=lffc
newunifact : L BP:=empty()
leadtest:=true --- the lc test with polCase has to be performed
int:L R:=empty()

-- New sets of integers are chosen to reduce the multivariate problem to
-- a univariate one, until we find twice the
-- same (and minimal) number of "univariate" factors:
-- the set smaller in modulo is chosen.
-- Note that there is no guarantee that this is the truth:
-- merely the closest approximation we have found!

while true repeat
    testp and #ltry>10 => return "failed"
lval := [ ran(range) for i in 1..nvar1]
member?(lval,ltry) => range:=2*range
ltry := cons(lval,ltry)
leadcomp1:=[retract eval(pol,lvar,lval) for pol in plist]
testp and or/
    [unit? epl for epl in leadcomp1] => range:=2*range
newm:BP:=completeEval(um,lvar,lval)
degum ^= degree newm or minimumDegree newm ^=0 => range:=2*range
lffc1:=content newm
newm:=(newm exquo lffc1)::BP
testp and leadtest and " polCase(lffc1*clc,#plist,leadcomp1)"
    => range:=2*range
degree(gcd [newm,differentiate(newm)])=0 => range:=2*range
luniv:=ufactor(newm)
lunivf:=factors luniv
lffc1:R:=retract(unit luniv)@R * lffc1
nf:= #lunivf

nf=0 or nf>nfatt => "next values" --- pretest failed ---
--- the univariate polynomial is irreducible ---
if nf=1 then leave (unifact:=[newm])

-- the new integer give the same number of factors
nfatt = nf =>
-- if this is the first univariate factorization with polCase=true
-- or if the last factorization has smaller norm and satisfies
-- polCase
if leadtest or
((localNorm unifact > localNorm \[ff.factor for ff in lunivf\])
 and (^testp or polCase(lffc1*clc,#plist,leadcomp1))) then
 unifact:=[uf.factor for uf in lunivf]
 int:=lval
 lffc:=lffc1
 if testp then leadcomp:=leadcomp1
 leave "foundit"

-- the first univariate factorization, inizialize
nfatt > degum =>
 unifact:=[uf.factor for uf in lunivf]
 lffc:=lffc1
 if testp then leadcomp:=leadcomp1
 int:=lval
 leadtest := false
 nfatt := nf

nfatt>nf => -- for the previous values there were more factors
if testp then leadtest:=~polCase(lffc*clc,#plist,leadcomp)
 else leadtest:= false
-- if polCase=true we can consider the univariate decomposition
if ~leadtest then
 unifact:=[uf.factor for uf in lunivf]
 lffc:=lffc1
 if testp then leadcomp:=leadcomp1
 int:=lval
 nfatt := nf
[cons(int,ltry),unifact,lffc,leadcomp]$Valuf

---- The polynomial has mindeg>0 ----
simplify(m:P,lvar:L OV,lmdeg:L NNI,ufactor:UFactor):MFinalFact ==
 factorlist:L MParFact:=[]
poll:P:= 1$P
for x in lvar repeat
 i := lmdeg.(position(x,lvar))
i=0 => "next value"
poll:=poll*monomial(1$P,x,i)
factorlist:=cons([x::P,i]$MParFact,factorlist)
m := (m exquo poll1):P
ground? m => [retract m,factorlist]$MFinalFact
flead:=mFactor(m,ufactor)
flead.factors:=append(factorlist,flead.factors)
flead

-- This is the key internal function
-- We now know that the polynomial is square-free etc.,
-- We use intChoose to find a set of integer values to reduce the
-- problem to univariate (and for efficiency, intChoose returns
-- the univariate factors).
-- In the case of a polynomial leading coefficient, we check that this
-- is consistent with leading coefficient determination (else try again)
-- We then lift the univariate factors to multivariate factors, and
-- return the result
intfact(um:USP,lvar: L OV,ldeg:L NNI,tleadpol:MFinalFact,
ltry:L L R,ufactor:UFactor) : L USP ==
  polcase:Boolean:= (not empty? tleadpol.factors)
  vfchoo:Valuf:=
    polcase =>
      leadpol:L P:=[ff.irr for ff in tleadpol.factors]
      check:=intChoose(um,lvar,tleadpol.contp,leadpol,ltry,ufactor)
      check case "failed" => return monicMfpol(um,lvar,ldeg,ufactor)
      check::Valuf
    intChoose(um,lvar,1,empty(),empty(),ufactor)::Valuf
  unifact:List BP := vfchoo.unvfact
  nfact=NNI := #unifact
  nfact=1 => [um]
  ltry:L L R := vfchoo.inval
  lval:L R := first ltry
  dd:= vfchoo.lu
  leadval:L R := empty()
  lpol:List P := empty()
  if polcase then
    leadval := vfchoo.complead
    distf := distFact(vfchoo.lu,unifact,tleadpol,leadval,lvar,lval)
    distf case "failed" =>
      return intfact(um,lvar,ldeg,tleadpol,ltry,ufactor)
    dist := distf :: LeadFact
      -- check the factorization of leading coefficient
    lpol:= dist.polfac
    dd := dist.correct
    unifact:=dist.corrfact
    if dd^=1 then
      if polcase then lpol := [unitCanonical lp for lp in lpol]
      --
      dd:=unitCanonical(dd)
      unifact := [dd * unif for unif in unifact]
      umd := unitNormal(dd).unit * ((dd**(nfact-1)::NNI)::P)*um
      else umd := um
      (ffin:=lifting(umd,lvar,unifact,lval,lpol,ldeg,mpmod))
      case "failed" => intfact(um,lvar,ldeg,tleadpol,ltry,ufactor)
PACKAGE INNMFACT INNERMULTFACT

factfin: L USP := ffin :: L USP
if dd == 1 then
    factfin := [primitivePart ff for ff in factfin]

---- m square-free, primitive, lc constant ----

mfconst(um:USP, lvar:L OV, ldeg:L NNI, ufactor:UFactor): L USP ==
    factfin := L USP := empty()
    empty? lvar =>
    lum := factors ufactor (map (ground, um) $UPCF2 (P, USP, R, BP))
    [map (coerce, uf.factor) $UPCF2 (R, BP, P, USP) for uf in lum]
    intfact (um, lvar, ldeg, [0, empty()]) $MFinalFact, empty(), ufactor)

monicize(um:USP, c:P): USP ==
    n := degree (um)
    ans := USP := monomial (1, n)
    n := (n - 1):: NonNegativeInteger
    prod := 1
    while (um := reductum (um)) ^= 0 repeat
        i := degree um
        lc := leadingCoefficient um
        prod := prod * c ** (n - (n := i)):: NonNegativeInteger
        ans := ans + monomial (prod * lc, i)
    ans

unmonicize(m:USP, c:P): USP == primitivePart m (monomial (c, 1))

--- m is square-free, primitive, lc is a polynomial ---

monicMfpol(um:USP, lvar:L OV, ldeg:L NNI, ufactor:UFactor): L USP ==
    l := leadingCoefficient um
    monpol := monicize (um, l)
    nldeg := degree (monpol, lvar)
    map (z1:USP):USP += unmonicize (z1, l),
    mfconst (monpol, lvar, nldeg, ufactor))

mfpol(um:USP, lvar:L OV, ldeg:L NNI, ufactor:UFactor): L USP ==
    R has Field =>
    monicMfpol (um, lvar, ldeg, ufactor)
    tleadpol := mFactor (leadingCoefficient um, ufactor)
    intfact (um, lvar, ldeg, tleadpol, [], ufactor)

mFactor(m:P, ufactor:UFactor) : MFinalFact ==
    ground? (m) => [retract (m), empty()] $MFinalFact
    lvar := L OV := variables m
    lcont := P
    if L USP
    flead := MFinalFact := [0, empty()] $MFinalFact
    factorlist := L MParFact := empty()

lmdeg := minimumDegree (m, lvar) ---- is the Mindeg > 0? ----
or/[\text{in}>0 \text{ for } n \text{ in } \text{lmdge}] \Rightarrow \text{ simplify}(m, l\text{var}, l\text{mdge}, u\text{factor})

sqfacs := \text{ squareFree } m
lcont := \text{ unit } sqfacs

\text{---- Factorize the content ----}
if \text{ ground? } lcont \text{ then } flead.\text{contp} := \text{ retract } lcont
else flead := \text{ mFactor}(lcont, u\text{factor})
factorlist := flead.\text{factors}

\text{---- Make the polynomial square-free ----}

sqqfact := \text{ factors } sqfacs

\text{--- Factorize the primitive square-free terms ---}
for fact in sqqfact repeat
ffactor := fact.\text{factor}
ffexp := fact.\text{exponent}
l\text{var} := \text{ variables } ffactor
x:OV := l\text{var}.\text{first}
l\text{deg} := \text{ degree}(ffactor, l\text{var})

\text{--- Is the polynomial linear in one of the variables ? ---}
if member?(1, l\text{deg}) =>
x:OV := l\text{var}.\text{position}(1, l\text{deg})
lcont := \text{ gcd coefficients}(\text{univariate}(ffactor, x))
ffactor := (ffactor \text{ exquo } lcont) \cdot P
factorlist := cons([[ffactor, ffexp] \cdot \text{MParFact}, \text{factorlist})
for lterm in \text{ mFactor}(lcont, u\text{factor}).\text{factors} repeat
factorlist := cons([lterm.\text{irr}, lterm.\text{pow} \cdot ffexp], factorlist)

varch := \text{varChoose}(ffactor, l\text{var}, l\text{deg})
um := varch.\text{npol}

x := l\text{var}.\text{first}
l\text{deg} := l\text{deg}.\text{rest}
l\text{var} := l\text{var}.\text{rest}
if varch.\text{nvar}.\text{first} = x then
l\text{var} := varch.\text{nvar}
x := l\text{var}.\text{first}
l\text{var} := l\text{var}.\text{rest}
pc := \text{ gcd } \text{ coefficients } um
if pc = 1 then
um := (um \text{ exquo } pc) \cdot \text{USP}
ffactor := \text{ multivariate}(um, x)
for lterm in \text{ mFactor}(pc, u\text{factor}).\text{factors} repeat
factorlist := cons([lterm.\text{irr}, lterm.\text{pow} \cdot ffexp], factorlist)
l\text{deg} := \text{ degree}(ffactor, l\text{var})
um := \text{ unitCanonical } um
if \text{ ground?}(\text{leadingCoefficient } um) \text{ then}
lf := mfconst(um, lvar, ldeg, ufactor)
else lf := mfpol(um, lvar, ldeg, ufactor)
auxfl := [[unitCanonical multivariate(lfp, x), ffexp] $MParFact for lfp in lf]
factorlist := append(factorlist, auxfl)
lcfacs := */[leadingCoefficient(f.iri) ** ((f.pow)::NNI) for f in factorlist]
[(leadingCoefficient(m) exquo lcfacs):: R, factorlist] $MFinalFact

factor(m: P, ufactor: UFactor): Factored P ==
flist := mFactor(m, ufactor)
(flist.contp):: P *
(*/[primeFactor(u.iri, u.pow) for u in flist.factors])
InnerNormalBasisFieldFunctions examples

InnerNormalBasisFieldFunctions(GF) (unexposed): This package has functions used by every normal basis finite field extension domain.

See Also:
  o )show InnerNormalBasisFieldFunctions

---

InnerNormalBasisFieldFunctions (INBFF)

Exports:

<table>
<thead>
<tr>
<th>basis</th>
<th>dAndcExp</th>
<th>expPot</th>
<th>index</th>
<th>inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup</td>
<td>minimalPolynomial</td>
<td>norm</td>
<td>normalElement</td>
<td>normal?</td>
</tr>
<tr>
<td>pol</td>
<td>qPot</td>
<td>random</td>
<td>repSq</td>
<td>setFieldInfo</td>
</tr>
<tr>
<td>trace</td>
<td>xn</td>
<td><strong>?</strong></td>
<td>???</td>
<td>??</td>
</tr>
</tbody>
</table>

--- package INBFF InnerNormalBasisFieldFunctions ---

)abbrev package INBFF InnerNormalBasisFieldFunctions
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated: 31 March 1991
++ References:
  ++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
  ++ D.R.Stinson: Some observations on parallel Algorithms for fast
  ++ exponentiation in GF(2^n), Siam J. Comp., Vol.19, No.4, pp.711-717,
  ++ August 1990
  ++ T.Itoh, S.Tsuji: A fast algorithm for computing multiplicative inverses
  ++ in GF(2^m) using normal bases, Inf. and Comp. 78, pp.171-177, 1988
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ InnerNormalBasisFieldFunctions(GF) (unexposed):
++ This package has functions used by
++ every normal basis finite field extension domain.

InnerNormalBasisFieldFunctions(GF): Exports == Implementation where
GF : FiniteFieldCategory       -- the ground field
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
I  ==> Integer
SI ==> SingleInteger
SUP ==> SparseUnivariatePolynomial
VGF ==> Vector GF
M  ==> Matrix
V  ==> Vector
L  ==> List
OUT ==> OutputForm
TERM ==> Record(value:GF,index:SI)
MM  ==> ModMonic(GF,SUP GF)

Exports == with

  setFieldInfo: (V L TERM,GF) -> Void
    ++ setFieldInfo(m,p) initializes the field arithmetic, where m is
    ++ the multiplication table and p is the respective normal element
    ++ of the ground field GF.
  random : PI -> VGF
    ++ random(n) creates a vector over the ground field with random entries.
  index : (PI,PI) -> VGF
    ++ index(n,m) is a index function for vectors of length n over
    ++ the ground field.
  pol : VGF -> SUP GF
    ++ pol(v) turns the vector \spad{[v_0,...,v_n]} into the polynomial
    ++ \spad{v_0+v_1*x+ ... + v_n*x^n}.
  xn : NNI -> SUP GF
    ++ xn(n) returns the polynomial \spad{x^n-1}.
  dAndcExp : (VGF,NNI,SI) -> VGF
    ++ dAndcExp(v,n,k) computes \spad{v^{n^k}} interpreting v as an element of
    ++ normal basis field. A divide and conquer algorithm similar to the
    ++ one from D.R.Stinson,
    ++ "Some observations on parallel Algorithms for fast exponentiation in
    ++ GF(2^n)", Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990
    ++ is used. Argument k is a parameter of this algorithm.
  repSq : (VGF,NNI) -> VGF
    ++ repSq(v,e) computes \spad{v^{e}} by repeated squaring,
    ++ interpreting v as an element of a normal basis field.
  expPot : (VGF,SI,SI) -> VGF
++ expPot(v,e,d) returns the sum from \( i = 0 \) to
++ \( e - 1 \) of \( v^{q^i \cdot d} \), interpreting
++ \( v \) as an element of a normal basis field and where \( q \) is
++ the size of the ground field.
++ Note that for a description of the algorithm, see
++ T. Itoh and S. Tsujii,
++ "A fast algorithm for computing multiplicative inverses in GF\( (2^m) \)
++ using normal bases",

qPot : (VGF,I) -> VGF
++ qPot(v,e) computes \( v^{q^e} \), interpreting \( v \) as an element of
++ normal basis field, \( q \) the size of the ground field.
++ This is done by a cyclic \( e \)-shift of the vector \( v \).

-- the semantic of the following functions is obvious from the finite field
-- context, for description see category FAXF

"**" : (VGF,I) -> VGF
++ x**n \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{**}{DivisionRing}}

"*" : (VGF,VGF) -> VGF
++ x*y \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{*}{SemiGroup}}

"/" : (VGF,VGF) -> VGF
++ x/y \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{/}{Field}}

norm : (VGF,PI) -> VGF
++ norm(x,n) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{norm}{FiniteAlgebraicExtensionField}}

trace : (VGF,PI) -> VGF
++ trace(x,n) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{trace}{FiniteAlgebraicExtensionField}}

inv : VGF -> VGF
++ inv x \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{inv}{DivisionRing}}

lookup : VGF -> PI
++ lookup(x) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{lookup}{Finite}}

normal? : VGF -> Boolean
++ normal?(x) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{normal?}{FiniteAlgebraicExtensionField}}

basis : PI -> V VGF
++ basis(n) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{basis}{FiniteAlgebraicExtensionField}}

normalElement : PI -> VGF
++ normalElement(n) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{normalElement}{FiniteAlgebraicExtensionField}}

minimalPolynomial : VGF -> SUP GF
++ minimalPolynomial(x) \texttt{undocumented{}}
++ See \texttt{axiomFunFrom{minimalPolynomial}{FiniteAlgebraicExtensionField}}
Implementation ==> add

-- global variables ===============================================================

sizeGF:NNI:=size()$GF
-- the size of the ground field

multTable:V L TERM:=new(1,nil()$(L TERM))$(V L TERM)
-- global variable containing the multiplication table

trGen:GF:=1$GF
-- controls the imbedding of the ground field

logq:List SI:=[0,10::SI,20::SI,30::SI,1,2,2,1,2,1,2,3,0,28::SI,...
-- logq.i is about 10*log2(i) for the values <12 which
-- can match sizeGF. It's used by "**"

expTable:L L SI:=[[],
[4::SI,12::SI,48::SI,16:SI,480::SI,160::SI,480::SI,0],
[8::SI,72::SI,432::SI,0],
[16::SI,216::SI,0],
[32::SI,480::SI,0],
[72::SI,0],[96::SI,0],[128::SI,0],[0],[200::SI,0]]
-- expT is used by "**" to optimize the parameter k
-- before calling dAndcExp(.,.,.,k)

-- functions =====================================================================

-- computes a**(-1) = a**((q**extDeg)-2)
-- see reference of function expPot
inv(a) ==
b:VGF:=qPot(expPot(a,(#a-1)::NNI::SI,1::SI)$$,1)$$
erg:VGF:=inv((a *$$ b).1 *$GF trGen)$GF *$VGF b

-- "**" decides which exponentiation algorithm will be used, in order to
-- get the fastest computation. If dAndcExp is used, it chooses the
-- optimal parameter k for that algorithm.
-- (a ** ex ==
e:NNI:=positiveRemainder(ex,sizeGF**((#a)::PI)-1)$I :: NNI
zero?(e)$NNI => new(#a, trGen)$VGF
-- one?(e)$NNI => copy(a)$VGF
(e = 1)$NNI => copy(a)$VGF
-- inGroundField?(a) => new(#a,((a.1*trGen) **$GF e))$VGF
e1:SI:=(length(e)$I)::SI
sizeGF >$I 11 =>
q1:SI:=(length(sizeGF)$I)::SI
logqe:SI:=(e1 quo$SI q1) +$SI 1$SI
10::SI * (logqe + sizeGF-2) > 15::SI * e1 =>
-- print("repeatedSquaring":::OUT)
CHAPTER 10. CHAPTER I

repSq(a,e)  
-- print("divAndConquer(a,e,1)":OUT)  
dAndcExp(a,e,1)  
logqe:SI:=((10::SI *$SI e1) quo$SI (logq.sizeGF)) +$SI 1$SI  
k:SI:=i$SI  
expT:List SI:=expTable.sizeGF  
while (logqe >= expT.k) and not zero? expT.k repeat k:=k +$SI 1$SI  
mult:I:=(sizeGF-1) *$I sizeGF **$I ((k-1)pretend NNI) +$I_  
((logqe +$SI k -$SI 1$SI) quo$SI k)::I -$I 2  
(10*mult) => (15 * (e1::I)) =>  
-- print("repeatedSquaring(a,e)":OUT)  
repSq(a,e)  
-- print(hconcat(["divAndConquer(a,e,"::OUT,k::OUT,")":OUT])$OUT)  
dAndcExp(a,e,k)  

-- computes a**e by repeated squaring  
repSq(b,e) ==  
an:=copy(b)$VGF  
-- one? e => a  
(e = 1) => a  
odd?(e)$I => a * repSq(a*a,(e quo 2) @ NNI)  
repSq(a*a,(e quo 2) @ NNI)  

-- computes a**e using the divide and conquer algorithm similar to the  
-- one from D.R. Stinson,  
-- "Some observations on parallel Algorithms for fast exponentiation in  
-- GF(2**n)", Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990  
dAndcExp(a,e,k) ==  
plist:List VGF:=[copy(a)$VGF]  
qk:I:=sizeGF***(k pretend NNI)  
for j in 2..(qk-1) repeat  
  if positiveRemainder(j,sizeGF)=0 then b:=qPot(plist.(j quo sizeGF),1)$$  
else b:=a *$$ last(plist)$VGF  
plist:=concat(plist,b)  
l:List NNI:=nil()  
ex:I:=e  
while not(ex = 0) repeat  
l:=concat(l,positiveRemainder(ex,qk) pretend NNI)  
ex:=ex quo qk  
if first(l)=0 then erg:VGF:=new(#a,trGen)$VGF  
else erg:VGF:=plist.(first(l))  
i:SI:=k  
for j in rest(l) repeat  
  if j=0 then erg:=$$ qPot(plist.j,i)$$  
i:=i+k  
erg  
a * b ==  
e:SI:=(#a)::SI  
erg:zero(#a)$VGF
for t in multTable.1 repeat
  for j in 1..e repeat
    y:=t.value -- didn't work without defining x and y
    x:=t.index
    k:SI:=addmod(x,j::SI,e)$SI +$SI 1$SI
    erg.k:=erg.k +$GF a.j *$GF b.j *$GF y
  for i in 1..e-1 repeat
    for j in i+1..e repeat
      for t in multTable.(j-i+1) repeat
        y:=t.value -- didn't work without defining x and y
        x:=t.index
        k:SI:=addmod(x,i::SI,e)$SI +$SI 1$SI
        erg.k:GF:=erg.k +$GF (a.i *$GF b.j +$GF a.j *$GF b.i) *$GF y
      erg

lookup(x) ==
  erg:1:=0
  for j in (#x)..1 by -1 repeat
    erg:=(erg * sizeGF) + (lookup(x.j)$GF rem sizeGF)
  erg=0 => (sizeGF**(#x)) :: PI
  erg :: PI

-- computes the norm of a over GF**d, d must devide extdeg
-- see reference of function expPot below
norm(a,d) ==
  dSI:=d::SI
  r:=divide((#a)::SI,dSI)
  not(r.remainder = 0) => error "norm: 2.arg must divide extdeg"
  expPot(a,r.quotient,dSI)$$

-- computes expPot(a,e,d) = sum form i=0 to e-1 over a**(q**id))
-- see T. Itoh and S. Tsujii,
-- "A fast algorithm for computing multiplicative inverses in GF(2^m)
-- using normal bases",
-- Information and Computation 78, pp.171-177, 1988
expPot(a,e,d) ==
  deg:SI:=(#a)::SI
  e=1 => copy(a)$VGF
  k2:SI:=d
  y:=copy(a)
  if bit?(e,0) then
    erg:=copy(y)
    qpot:SI:=k2
  else
    erg:=new(#a,inv(trGen)$GF)$VGF
    qpot:SI:=0
  for k in 1..length(e) repeat
    y:= y *$*$ qPot(y,k2)
    k2:=addmod(k2,k2,deg)$SI
  if bit?(e,k) then
\[\text{erg} := \text{erg} \ast \text{qPot}(y, \text{qpot})\]
\[\text{qpot} := \text{addmod}(\text{qpot}, k2, \text{deg})\]$
\text{SI}$
\[\text{erg}\]

-- computes $\text{qPot}(a, n) = a^{\phi(n)}$, $q=$size of GF

\[\text{qPot}(a, n) ==\]
\[\text{ei} := (\#a) :: \text{SI}\]
\[m : \text{SI} := \text{positiveRemainder}(n : \text{SI}, \text{ei})\]$
\text{SI}$
\[\text{zero?} (m) \Rightarrow a\]
\[\text{ei} := \text{zero} (\#a) \]$
\text{VGF}$
\[\text{for} \ i \ \text{in} \ m+1..\text{ei} \ \text{repeat} \ \text{el}.i := \text{a}.(i-m)\]
\[\text{for} \ i \ \text{in} \ 1..m \ \text{repeat} \ \text{el}.i := \text{a}.(\text{ei}+i-m)\]
\[\text{ei}\]

\[\text{trace}(a, d) == \]
\[dS := d :: \text{SI}\]
\[r := \text{divide}((\#a) :: \text{SI}, dS)\]$
\text{SI}$
\[\text{not}(r.\text{remainder} = 0) \Rightarrow \text{error} \ "\text{trace}: 2.\text{arg must divide extdeg}\"
\[v := \text{copy}(a.(1..dS)) \]$
\text{VGF}$
\[sS := r.\text{quotient}\]
\[\text{for} \ i \ \text{in} \ 1..dS \ \text{repeat} \]
\[\text{for} \ j \ \text{in} \ 1..sS-1 \ \text{repeat} \]
\[v.i := v.i + a.(i+j :: dS)\]
\[v\]

\[\text{random}(n) == \]
\[v := \text{zero}(n) \]$
\text{VGF}$
\[\text{for} \ i \ \text{in} \ 1..n \ \text{repeat} \ v.i := \text{random}() \]$
\text{GF}$
\[v\]

\[\text{xn}(m) == \text{monomial}(1, m) \text{(SUP GF) - 1)(SUP GF)\]

\[\text{normal?}(x) == \]
\[\text{gcd} (\text{xn}(\#x), \text{pol}(x)) \text{(SUP GF) = 1 \Rightarrow true\ true}\]
\[\text{false}\]

\[x : \text{VGF} / y : \text{VGF} == x \ast \text{inv}(y)\]

\[\text{setFieldInfo}(m, n) == \]
\[\text{multiTable} := m\]
\[\text{trGen} := n\]
\[\text{void}() \]$
\text{Void}\]

\[\text{minimalPolynomial}(x) == \]
\[d := \#x\]
\[y := \text{new}(\#x, \text{inv}(\text{trGen}) \]$
\text{VGF}$
\[\text{m} := \text{zero}(d, d+1) \]$
\text{M GF}$
\[\text{for} \ i \ \text{in} \ 1..d+1 \ \text{repeat} \]

dy:=#y
for j in 1..dy repeat
  for k in 0..(dx quo dy)-1 repeat
    qsetelt_!(m,j+k*dy,i,y.j)$(M GF)
y:=y *$$ x
v:=first nullSpace(m)$(M GF)
pol(v)$$

basis(n) ==
  bas:>(V VGF):=new(n,zero(n)$VGF)$(V VGF)
  for i in 1..n repeat
    uniti:=zero(n)$VGF
    qsetelt_!(uniti,i,1$GF)$VGF
    qsetelt_!(bas,i,uniti)$(V VGF)
basis

normalElement(n) ==
  v:=zero(n)$VGF
  qsetelt_!(v,1,1$GF)
v
-- normalElement(n) == index(n,1)$$

index(degm,n) ==
  m:I:=n rem$I (sizeGF ** degm)
  erg:=zero(degm)$VGF
  for j in 1..degm repeat
    erg.j:=index((sizeGF+(m rem sizeGF)) pretend PI)$GF
  m:=m quo sizeGF
  erg

pol(x) ==
  +/[monomial(x.i,(i-1)::NNI)$(SUP GF) for i in 1..(#x)::I]

—— INBFF.dotabb ——

"INBFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INBFF"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"INBFF" -> "IVECTOR"

——

package INEP InnerNumericEigenPackage
CHAPTER 10. CHAPTER I

— InnerNumericEigenPackage.input —

)set break resume
)sys rm -f InnerNumericEigenPackage.output
)spool InnerNumericEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerNumericEigenPackage
--E 1

)spool
)lisp (bye)

— InnerNumericEigenPackage.help —

====================================================================
InnerNumericEigenPackage examples
====================================================================

This package is the inner package to be used by NumericRealEigenPackage
and NumericComplexEigenPackage for the computation of numeric
eigenvalues and eigenvectors.

See Also:
o )show InnerNumericEigenPackage

InnerNumericEigenPackage (INEP)
Exports:
charpol innerEigenvectors solve1

--- package INEP InnerNumericEigenPackage ---

)abbrev package INEP InnerNumericEigenPackage
++ Author:P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: Spring 1991
++ Description:
++ This package is the inner package to be used by NumericRealEigenPackage
++ and NumericComplexEigenPackage for the computation of numeric
++ eigenvalues and eigenvectors.

InnerNumericEigenPackage(K,F,Par) : C == T
where
  F : Field -- this is the field where the answer will be
   -- for dealing with the complex case
  K : Field -- type of the input
  Par : Join(Field,OrderedRing) -- it will be NF or RN

SE ==> Symbol()
RN ==> Fraction Integer
I ==> Integer
NF ==> Float
CF ==> Complex Float
GRN ==> Complex RN
GI ==> Complex Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
MRN ==> Matrix RN

MK ==> Matrix K
PK ==> Polynomial K
MF ==> Matrix F
SUK ==> SparseUnivariatePolynomial K
SUF ==> SparseUnivariatePolynomial F
SUP ==> SparseUnivariatePolynomial
MSUK ==> Matrix SUK

PEigenForm ==> Record(algpol:SUK,almult:Integer,poleigen:List(MSUK))

outForm ==> Record(outval:F,outmult:Integer,outvect:List MF)

IntForm ==> Union(outForm,PEigenForm)

UFactor ==> (SUK -> Factored SUK)

C == with
1140

CHAPTER 10. CHAPTER I

charpol : MK -> SUK
  ++ charpol(m) computes the characteristic polynomial of a matrix
  ++ m with entries in K.
  ++ This function returns a polynomial
  ++ over K, while the general one (that is in EiegenPackage) returns
  ++ Fraction P K

solve1 : (SUK, Par) -> List F
  ++ solve1(pol, eps) finds the roots of the univariate polynomial
  ++ polynomial pol to precision eps. If K is \spad{Fraction Integer}
  ++ then only the real roots are returned, if K is
  ++ \spad{Complex Fraction Integer} then all roots are found.

innerEigenvectors : (MK, Par, UFactor) -> List(outForm)
  ++ innerEigenvectors(m,eps,factor) computes explicitly
  ++ the eigenvalues and the correspondent eigenvectors
  ++ of the matrix m. The parameter eps determines the type of
  ++ the output, factor is the univariate factorizer to be used
  ++ to reduce the characteristic polynomial into irreducible factors.

T == add

numeric(r:K):F ==
  K is RN =>
    F is NF => convert(r)$RN
    F is RN => r
    F is CF => r :: RN :: CF
    F is GRN => r::RN::GRN
  K is GRN =>
    F is GRN => r
    F is CF => convert(convert r)
  error "unsupported coefficient type"

---- next functions neeeded for defining ModularField -----

monicize(f:SUK) : SUK ==
  (a:=leadingCoefficient f) =1 => f
  inv(a)*f

reduction(u:SUK,p:SUK):SUK == u rem p

merge(p:SUK,q:SUK):Union(SUK,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

  val:=extendedEuclidean(v,p,u)
  val case "failed" => "failed"
val.coef1

---- eval a vector of F in a radical expression ----
evalvect(vect:MSUK,alg:F) : MF ==
n:=nrows vect
w:=zero(n,1)$MF
for i in 1..n repeat
  polf:=map(numeric,
    vect(i,1))$UnivariatePolynomialCategoryFunctions2(K,SUK,F,SUF)
v:F:=elt(polf,alg)
setelt(w,i,1,v)
w

---- internal function for the computation of eigenvectors ----
inteigen(A:MK,p:SUK,fact:UFactor) : List(IntForm) ==
dimA:NNI:= nrows A
MM:=ModularField(SUK,SUK,reduction,merge,exactquo)
AM:=Matrix(MM)
lff:=factors fact(p)
res: List IntForm :=[]
lr : List MF:=[]
for ff in lff repeat
  pol:SUK:= ff.factor
  if (degree pol)=1 then
    alpha:K:=-coefficient(pol,0)/leadingCoefficient pol
    -- compute the eigenvectors, rational case
    B1:MK := zero(dimA,dimA)$MK
    for i in 1..dimA repeat
      for j in 1..dimA repeat B1(i,j):=A(i,j)
      B1(i,i):= B1(i,i) - alpha
    lr:=[]
    for vecr in nullSpace B1 repeat
      wf:MF:=zero(dimA,1)
      for i in 1..dimA repeat wf(i,1):=numeric vecr.i
      lr:=cons(wf,lr)
    res:=cons([[numeric alpha,ff.exponent,lr]$outForm,res])
  else
    ppol:=monicize pol
    alg:MM:= reduce(monomial(1,1),ppol)
    B:AM:= zero(dimA,dimA)$AM
    for i in 1..dimA repeat
      for j in 1..dimA repeat B(i,j):=reduce(A(i,j) ::SUK,ppol)
      B(i,i):=B(i,i) - alg
    sln2:=nullSpace B
    soln:List MSUK :=[]
    for vec in sln2 repeat
      wk:MSUK:=zero(dimA,1)
      for i in 1..dimA repeat wk(i,1):=numeric vec.i
      soln:=cons(wk,soln)
    res:=cons([[ff.factor,ff.exponent,soln]$PEigenForm,
res)

if K is RN then
  solve1(up:SUK, eps:Par) : List(F) ==
  denom := "lcm"/[denom(c::RN) for c in coefficients up]
  up:=denom*up
  upi:=map(numer,up)
  \$UnivariatePolynomialCategoryFunctions2(RN,SUP RN,I,SUP I)
  innerSolve1(upi, eps)$InnerNumericFloatSolvePackage(I,F,Par)
else if K is GRN then
  solve1(up:SUK, eps:Par) : List(F) ==
  denom := "lcm"/[lcm(denom real(c::GRN), denom imag(c::GRN))
  for c in coefficients up]
  up:=denom*up
  upgi := map((c:GRN):GI+->complex(numer(real c), numer(imag c)),up)_
  \$UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,GI,SUP GI)
  innerSolve1(upgi, eps)$InnerNumericFloatSolvePackage(GI,F,Par)
else error "unsupported matrix type"

---- the real eigenvectors expressed as floats ----

innerEigenvectors(A:MK,eps:Par,fact:UFactor) : List outForm ==
pol:= charpol A
sln1:List(IntForm):=inteigen(A,pol,fact)
n:=nrows A
sln:List(outForm):=[]
for lev in sln1 repeat
  lev case outForm => sln:=cons(lev,sln)
  leva:=lev::PEigenForm
  lval:List(F):= solve1(leva.algpol,eps)
  lvect:=leva.poleigen
  lmult:=leva.almult
  for alg in lval repeat
    nsl:=[alg,lmult, [evalvect(ep,alg) for ep in lvect]]$outForm
    sln:=cons(nsl,sln)
  sln

charpol(A:MK) : SUK ==
dimA :PI := (nrows A):PI
dimA "= ncols A => error " The matrix is not square"
B:Matrix SUK := zero(dimA,dimA)
for i in 1..dimA repeat
  for j in 1..dimA repeat  \ B(i, j) := A(i, j) :: SUK
  B(i, i) := B(i, i) - monomial(1, 1)$SUK
determinant B
package INFSP InnerNumericFloatSolvePackage

--- InnerNumericFloatSolvePackage.input ---

)set break resume
)sys rm -f InnerNumericFloatSolvePackage.output
)spool InnerNumericFloatSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerNumericFloatSolvePackage
--E 1

)spool
)lisp (bye)

--- InnerNumericFloatSolvePackage.help ---

====================================================================
InnerNumericFloatSolvePackage examples
====================================================================

This is an internal package for computing approximate solutions to systems of polynomial equations. The parameter K specifies the coefficient field of the input polynomials and must be either Fraction(Integer) or Complex(Fraction Integer).

The parameter F specifies where the solutions must lie and can be one of the following: Float, Fraction(Integer), Complex(Float), or Complex(Fraction Integer). The last parameter specifies the type of the precision operand and must be either Fraction(Integer) or Float.
See Also:
- \texttt{show InnerNumericFloatSolvePackage}

---

**InnerNumericFloatSolvePackage (INFSP)**

Exports:
- \texttt{innerSolve}
- \texttt{innerSolve1}
- \texttt{makeEq}

---

\texttt{InnerNumericFloatSolvePackage(K,F,Par): Cat == Cap where}

\begin{align*}
F & : \text{Field} \quad \text{-- this is the field where the answer will be} \\
K & : \text{GcdDomain} \quad \text{-- type of the input} \\
Par & : \text{Join(Field, OrderedRing)} \quad \text{-- it will be NF or RN}
\end{align*}
I ==> Integer
NNI ==> NonNegativeInteger
P ==> Polynomial
EQ ==> Equation
L ==> List
SUP ==> SparseUnivariatePolynomial
RN ==> Fraction Integer
NF ==> Float
CF ==> Complex Float
GI ==> Complex Integer
GRN ==> Complex RN
SE ==> Symbol
RFI ==> Fraction P I

Cat == with

innerSolve1 : (SUP K,Par) -> L F
++ innerSolve1(up,eps) returns the list of the zeros
++ of the univariate polynomial up with precision eps.
innerSolve1 : (P K,Par) -> L F
++ innerSolve1(p,eps) returns the list of the zeros
++ of the polynomial p with precision eps.
innerSolve : (L P K,L P K,L SE,Par) -> L L F
++ innerSolve(lnum,lden,lvar,eps) returns a list of
++ solutions of the system of polynomials lnum, with
++ the side condition that none of the members of lden
++ vanish identically on any solution. Each solution
++ is expressed as a list corresponding to the list of
++ variables in lvar and with precision specified by eps.
makeEq : (L F,L SE) -> L EQ P F
++ makeEq(lsol,lvar) returns a list of equations formed
++ by corresponding members of lvar and lsol.

Cap == add

------ Local Functions ------

isGeneric? : (L P K,L SE) -> Boolean
evaluate : (P K,SE,SE,F) -> F
numeric : K -> F
oldCoord : (L F,L I) -> L F
findGenZeros : (L P K,L SE,Par) -> L L F
failPolSolve : (L P K,L SE) -> Union(L L P K,"failed")

numeric(r:K):F ==
  K is I =>
  F is Float => r::I::Float
  F is RN   => r::I::RN
  F is CF   => r::I::CF
  F is GRN  => r::I::GRN
K is GI =>
  \( gr:GI := r::GI \)

F is GRN => \( \text{complex(real}(gr)::RN,\text{imag}(gr)::RN)\)$GRN
F is CF => convert(gr)
error "case not handled"

-- construct the equation
makeEq(nres:L F,lv:L SE) : L EQ P F ==
[equation(x::(P F),r::(P F)) for x in lv for r in nres]

  rpp:=map(numeric,pol)$PolynomialFunctions2(K,F)
  rpp := eval(rpp,zvar,z)
  upol:=univariate(rpp,xvar)
  retract(-coefficient(upol,0))/retract(leadingCoefficient upol)

myConvert(eps:Par) : RN ==
  Par is RN => eps
  Par is NF => retract(eps)$NF

innerSolve1(pol:P K,eps:Par) : L F == innerSolve1(univariate pol,eps)

innerSolve1(upol:SUP K,eps:Par) : L F ==
  K is GI and (Par is RN or Par is NF) =>
  (complexZeros(upol,
    eps)$ComplexRootPackage(SUP K,Par)) pretend L(F)
  K is I =>
  F is Float =>
  z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
  [convert((1/2)*(x.left+x.right))@Float for x in z] pretend L(F)
  F is RN =>
  z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
  [(1/2)*(x.left + x.right) for x in z] pretend L(F)
error "improper arguments to INFSP"
error "improper arguments to INFSP"

-- find the zeros of components in "generic" position --
findGenZeros(lp:L P K,rlvar:L SE,eps:Par) : L L F ==
  rlp:=reverse lp
  f:=rlp.first
  zvar:= rlvar.first
  rlp:=rlp.rest
  lz:=innerSolve1(f,eps)
  [reverse cons(z,[evaluate(pol,xvar,zvar,z) for pol in rlp
    for xvar in rlvar.rest]) for z in lz]

-- convert to the old coordinates --
oldCoord(numres:L F,lval:L I) : L F ==
rnumres := reverse numres
rnumres.first := rnumres.first +
(+/[n*nr for n in lval for nr in rnumres.rest])
reverse rnumres

-- real zeros of a system of 2 polynomials lp (incomplete)
mainvar := first lv
up1 := univariate(lp.1, mainvar)
up2 := univariate(lp.2, mainvar)
vec := subresultantVector(up1,up2)$SubResultantPackage(P K, SUP P K)
p0 := primitivePart multivariate(vec.0, mainvar)
p1 := primitivePart(multivariate(vec.1, mainvar), mainvar)
zero? p1 or
  gcd(p0, leadingCoefficient(univariate(p1,mainvar))) ^= 1 =>
    innerSolve(cons(0, lp), empty(), lv, eps)
findGenZeros([p1, p0], reverse lv, eps)

-- real zeros of the system of polynomial lp --
  -- empty?(ld) and (#lv = 2) and (# lp = 2) => innerSolve2(lp, lv, eps)
  lnp := [pToDmp(p)$PolToPol(lv, K) for p in lp]
  OV := OrderedVariableList(lv)
  lvv := [variable(vv)::OV for vv in lv]
  DP := DirectProduct(#lv, NonNegativeInteger)
  dmp := DistributedMultivariatePolynomial(lv, K)
lq := dmp: []
  if ld = [] then
    lq := [(pToDmp(q1)$PolToPol(lv, K)) pretend dmp for q1 in ld]
  partRes := groebSolve(lnp, lvv)$GroebnerSolve(lv, K, K) pretend (L L dmp)
  partRes = list [] => []
  -- remove components where denominators vanish
  if lq = [] then
    gb := GroebnerInternalPackage(K, DirectProduct(#lv, NNI), OV, dmp)
    partRes := [pr for pr in partRes |
      and/[[(redPol(fq, pr pretend List(dmp))$gb) ^= 0 for fq in lq]]
  -- select the components in "generic" form
  rlv := reverse lv
  rrlvv := rest reverse lvv
  listGen := L L dmp: []
  for res in partRes repeat
    res1 := rest reverse res
    "and"/[("max"/degree(f, rrlvv)) = 1 for f in res1] =>
      listGen := concat(res pretend (L dmp), listGen)
  result := L L F := []
  if listGen = [] then
    listG := L L P K:
CHAPTER 10. CHAPTER I

[[dmpToP(pf)$PolToPol(lv,K) for pf in pr] for pr in listGen] result:=
"append"/[findGenZeros(res,rlv,eps) for res in listG]
for gres in listGen repeat
partRes:=delete(partRes,position(gres,partRes))
-- adjust the non-generic components
for gres in partRes repeat
  genRecord := genericPosition(gres,lvv)$GroebnerSolve(lv,K,K)
  lgen := genRecord.dpolys
  lval := genRecord.coords
  lgen1:=[dmpToP(pf)$PolToPol(lv,K) for pf in lgen]
  lris:=findGenZeros(lgen1,rlv,eps)
  result:= append([oldCoord(r,lval) for r in lris],result)
result

——

— INFSP.dotabb —

"INFSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFSP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"INFSP" -> "COMPCAT"

——

package INPSIGN InnerPolySign

— InnerPolySign.input —

)set break resume
/sys rm -f InnerPolySign.output
/spool InnerPolySign.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerPolySign
--E 1

)spool
)lisp (bye)

——
--- InnerPolySign.help ---

====================================================================
InnerPolySign examples
====================================================================

Find the sign of a polynomial around a point or infinity.

See Also:
o )show InnerPolySign

---

InnerPolySign (INPSIGN)

Exports:
signAround

--- package INPSIGN InnerPolySign ---

)abbrev package INPSIGN InnerPolySign
++ Author: Manuel Bronstein
++ Date Created: 23 Aug 1989
++ Date Last Updated: 19 Feb 1990
++ Description:
++ Find the sign of a polynomial around a point or infinity.

InnerPolySign(R, UP):Exports == Implementation where
R : Ring
UP: UnivariatePolynomialCategory R
U ==> Union(Integer, "failed")
Exports ==> with
  signAround: (UP, Integer, R -> U) -> U
    ++ signAround(u,i,f) \ undocumented
  signAround: (UP, R, Integer, R -> U) -> U
    ++ signAround(u,r,i,f) \ undocumented
  signAround: (UP, R, R -> U) -> U
    ++ signAround(u,r,f) \ undocumented

Implementation ==> add
  signAround(p:UP, x:R, rsign:R -> U) ==
    (ur := signAround(p, x, 1, rsign)) case "failed" => "failed"
    (ul := signAround(p, x, -1, rsign)) case "failed" => "failed"
    (ur::Integer) = (ul::Integer) => ur
    "failed"

  signAround(p, x, dir, rsign) ==
    zero? p => 0
    zero?(r := p x) =>
      (u := signAround(differentiate p, x, dir, rsign)) case "failed"
        => "failed"
      dir * u::Integer
      rsign r

  signAround(p:UP, dir:Integer, rsign:R -> U) ==
    zero? p => 0
    (u := rsign leadingCoefficient p) case "failed" => "failed"
    (dir > 0) or (even? degree p) => u::Integer
    - (u::Integer)

——

— INPSIGN.dotabb —

"INPSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPSIGN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INPSIGN" -> "PFECAT"

——

package ISUMP InnerPolySum

— InnerPolySum.input —

)set break resume
InnerPolySum (ISUMP)

Exports:
sum
package ISUMP InnerPolySum

)abbrev package ISUMP InnerPolySum
++ Author: SMW
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for the summation packages of polynomials

InnerPolySum(E, V, R, P): Exports == Impl where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: IntegralDomain
  P: PolynomialCategory(R, E, V)

  Z ==> Integer
  Q ==> Fraction Z
  SUP ==> SparseUnivariatePolynomial

Exports == with
  sum: (P, V, Segment P) -> Record(num:P, den:Z)
    ++ sum(p(n), n = a..b) returns \spad{p(a) + p(a+1) + ... + p(b)}.
  sum: (P, V) -> Record(num:P, den: Z)
    ++ sum(p(n), n) returns \spad{P(n)},
    ++ the indefinite sum of \spad{p(n)} with respect to
    ++ upward difference on n, i.e. \spad{P(n+1) - P(n) = a(n)};

Impl ==> add
  import PolynomialNumberTheoryFunctions()
  import UnivariatePolynomialCommonDenominator(Z, Q, SUP Q)

  pmul: (P, SUP Q) -> Record(num:SUP P, den:Z)
  pmul(c, p) ==
    pn := (rec := splitDenominator p).num
    [map(x +-> numer(x) * c, pn), rec.den]
    $SparseUnivariatePolynomialFunctions2(Q, P), rec.den]

  sum(p, v, s) ==
    indef := sum(p, v)
    [eval(indef.num, v, 1 + hi s) - eval(indef.num, v, lo s),
     indef.den]

  sum(p, v) ==
    up := univariate(p, v)
    lp := nil()$List(SUP P)
    ld := nil()$List(Z)
    while up ^= 0 repeat
      ud := degree up; uc := leadingCoefficient up
      up := reductum up
rec := pmul(uc, 1 / (ud+1) * bernoulli(ud+1))
lp := concat(rec.num, lp)
ld := concat(rec.den, ld)
d := lcm ld
vp := +/[[(d exquo di)::Z * pi for di in ld for pi in lp]
[multivariate(vp, v), d]

package ITRIGMNP InnerTrigonometricManipulations

-- InnerTrigonometricManipulations.input --

)set break resume
)sys rm -f InnerTrigonometricManipulations.output
)spool InnerTrigonometricManipulations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerTrigonometricManipulations
--E 1

)spool
)lisp (bye)

-- InnerTrigonometricManipulations.help --

====================================================================
InnerTrigonometricManipulations examples
====================================================================
This package provides transformations from trigonometric functions to exponentials and logarithms, and back.

F and FG should be the same type of function space.

See Also:
  o )show InnerTrigonometricManipulations

InnerTrigonometricManipulations (ITRIGMNP)

Exports:
  F2FG  FG2F  GF2FG  explogs2trigs  trigs2explogs

— package ITRIGMNP InnerTrigonometricManipulations —

)abbrev package ITRIGMNP InnerTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 9 October 1993
++ Description:
++ This package provides transformations from trigonometric functions to exponentials and logarithms, and back.
++ F and FG should be the same type of function space.

InnerTrigonometricManipulations(R,F,FG): Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet)
  F : Join(FunctionSpace R, RadicalCategory, TranscendentalFunctionCategory)
  FG : Join(FunctionSpace Complex R, RadicalCategory, TranscendentalFunctionCategory)
Z ==> Integer
SY ==> Symbol
OP ==> BasicOperator
GR ==> Complex R
GF ==> Complex F
KG ==> Kernel FG
PG ==> SparseMultivariatePolynomial(GR, KG)
UP ==> SparseUnivariatePolynomial PG
NTHR ==> "nthRoot": SY

Exports ==> with
GF2FG : GF -> FG
++ GF2FG(a + i b) returns \spad{a + i b} viewed as a function with
++ the \spad{i} pushed down into the coefficient domain.
FG2F : F -> F
++ FG2F(a + i b) returns \spad{a + sqrt(-1) b}.
F2FG : F -> FG
++ F2FG(a + sqrt(-1) b) returns \spad{a + i b}.

explogs2trigs: FG -> GF
++ explogs2trigs(f) rewrites all the complex logs and
++ exponentials appearing in \spad{f} in terms of trigonometric
++ functions.
trigs2explogs: (FG, List KG, List SY) -> FG
++ trigs2explogs(f, [k1,...,kn], [x1,...,xm]) rewrites
++ all the trigonometric functions appearing in \spad{f} and involving
++ one of the \spad{xi}’s in terms of complex logarithms and
++ exponentials. A kernel of the form \spad{tan(u)} is expressed
++ using \spad{exp(u)**2} if it is one of the \spad{ki}’s, in terms of
++ \spad{exp(2*u)} otherwise.

Implementation ==> add
ker2explogs: (KG, List KG, List SY) -> FG
smp2explogs: (PG, List KG, List SY) -> FG
supexp : (UP, GF, GF, Z) -> GF
GR2GF : GR -> GF
GR2F : GR -> F
KG2F : KG -> F
PG2F : PG -> F
ker2trigs : (OP, List GF) -> GF
smp2trigs : PG -> GF
sup2trigs : (UP, GF) -> GF

nth := R has RetractableTo(Integer) and F has RadicalCategory

GR2F g == real(g)::F + sqrt(-(1::F)) * imag(g)::F
KG2F k == map(FG2F, k)$ExpressionSpaceFunctions2(FG, F)
FG2F f == (PG2F numer f) / (PG2F denom f)
F2FG f == map(x -> x::GR, f)$FunctionSpaceFunctions2(R, F, GR, FG)
GF2FG f == (F2FG real f) + complex(0, 1)$GR :: FG * F2FG imag f
GR2GF gr == complex(real(gr)::F, imag(gr)::F)
CHAPTER 10. CHAPTER I

-- This expects the argument to have only tan and atans left.
-- Does a half-angle correction if k is not in the initial kernel list.
ker2explogs(k, 1, lx) ==
  empty?([v for v in variables(kf := k::FG) | member?(v, lx)]$List(SY)) => kf
  empty?(args := [trigs2explogs(a, 1, lx)
    for a in argument k]$List(FG)) => kf
  im := complex(0, 1)$GR :: FG
  z := first args
  is?(k, "tan"::Symbol) =>
    e := (member?(k, l) => exp(im * z) ** 2; exp(2 * im * z))
    - im * (e - 1) /$FG (e + 1)
  is?(k, "atan"::Symbol) =>
    im * log((1 -$FG im *$FG z)/$FG (1 +$FG im *$FG z))$FG / (2::FG)
  (operator k) args

trigs2explogs(f, 1, lx) ==
  smp2explogs(numer f, l, lx) / smp2explogs(denom f, l, lx)

-- return op(arg) as f + %i g
-- op is already an operator with semantics over R, not GR
ker2trigs(op, arg) ==
  "and"/[zero? imag x for x in arg] =>
    compler(op [real x for x in arg]$List(F), 0)
  a := first arg
  is?(op, "exp"::Symbol) => exp a
  is?(op, "log"::Symbol) => log a
  is?(op, "sin"::Symbol) => sin a
  is?(op, "cos"::Symbol) => cos a
  is?(op, "tan"::Symbol) => tan a
  is?(op, "cot"::Symbol) => cot a
  is?(op, "sec"::Symbol) => sec a
  is?(op, "csc"::Symbol) => csc a
  is?(op, "asin"::Symbol) => asin a
  is?(op, "acos"::Symbol) => acos a
  is?(op, "atan"::Symbol) => atan a
  is?(op, "acot"::Symbol) => acot a
  is?(op, "asec"::Symbol) => asec a
  is?(op, "acsc"::Symbol) => acsc a
  is?(op, "sinh"::Symbol) => sinh a
  is?(op, "cosh"::Symbol) => cosh a
  is?(op, "tanh"::Symbol) => tanh a
  is?(op, "coth"::Symbol) => coth a
  is?(op, "sech"::Symbol) => sech a
  is?(op, "csch"::Symbol) => csch a
  is?(op, "asinh"::Symbol) => asinh a
  is?(op, "acosh"::Symbol) => acosh a
  is?(op, "atanh"::Symbol) => atanh a
  is?(op, "acoth"::Symbol) => acoth a
is?(op, "asech"::Symbol) => asech a
is?(op, "acsch"::Symbol) => acsch a
is?(op, "abs"::Symbol) => sqrt(norm a)::GF
nth and is?(op, NTHR) => nthRoot(a, retract(second arg)@Z)
error "ker2trigs: cannot convert kernel to gaussian function"

sup2trigs(p, f) ==
  map(smp2trigs, p)$SparseUnivariatePolynomialFunctions2(PG, GF) f

smp2trigs p ==
  map(x +-> explogs2trigs(x::FG),GR2GF, p)
    $PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, GF)

explogs2trigs f ==
  (m := mainKernel f) case "failed" =>
    GR2GF(retract(numer f){GR} / GR2GF(retract(denom f){GR})
  op := operator(operator(k := m::KG))$F
  arg := [explogs2trigs x for x in argument k]
  num := univariate(numer f, k)
  den := univariate(denom f, k)
  is?(op, "exp"::Symbol) =>
    e := exp real first arg
    y := imag first arg
    g := complex(e * cos y, e * sin y)$GF
    gi := complex(cos(y) / e, - sin(y) / e)$GF
    supexp(num,g,gi,b := (degree num)::Z quo 2)/supexp(den,g,gi,b)
    sup2trigs(num, g := ker2trigs(op, arg)) / sup2trigs(den, g)

supexp(p, f1, f2, bse) ==
  ans:GF := 0
  while p ^= 0 repeat
    g := explogs2trigs(leadingCoefficient(p)::FG)
    if ((d := degree(p)::Z - bse) >= 0) then
      ans := ans + g * f1 ** d
    else ans := ans + g * f2 ** (-d)
    p := reductum p
    ans

PG2F p ==
  map(KG2F, GR2F, p)$PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, F)

smp2explogs(p, 1, lx) ==
  map(x +-> ker2explogs(x, 1, lx), y +-> y::FG, p)
    $PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, FG)
package INFORM1 InputFormFunctions1

--- InputFormFunctions1.input ---

)set break resume
)sys rm -f InputFormFunctions1.output
)spool InputFormFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InputFormFunctions1
--E 1

)spool
)lisp (bye)

--- InputFormFunctions1.help ---

====================================================================
InputFormFunctions1 examples
====================================================================

Tools for manipulating input forms.

See Also:
  o )show InputFormFunctions1
InputFormFunctions1 (INFORM1)

Exports:
   interpret  packageCall

--- package INFORM1 InputFormFunctions1 ---

)abbrev package INFORM1 InputFormFunctions1
--$boot $noSubsumption := false
++ Author: Manuel Bronstein
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for manipulating input forms.

InputFormFunctions1(R:Type):with
   packageCall: Symbol -> InputForm
      ++ packageCall(f) returns the input form corresponding to f$R.
   interpret : InputForm -> R
      ++ interpret(f) passes f to the interpreter, and transforms
      ++ the result into an object of type R.
== add
   Rname := devaluate(R)$Lisp :: InputForm

   packageCall name ==
      convert([convert("elt":Symbol), Rname, convert name]$List(InputForm))@InputForm

   interpret form ==
      retract(interpret(convert([convert("@":Symbol), form, Rname]$List(InputForm))@InputForm)$InputForm)$AnyFunctions1(R)
package INTERGB InterfaceGroebnerPackage

-- InterfaceGroebnerPackage.input --

)set break resume
)sys rm -f InterfaceGroebnerPackage.output
)spool InterfaceGroebnerPackage.output
)set message test on
)set message auto off
)clear all
--$ 1 of 1
)show InterfaceGroebnerPackage
--R
--R InterfaceGroebnerPackage(K: Field,symb: List(Symbol),E: OrderedAbelianMonoidSup,OV: OrderedSet,R: PolynomialCategory(K,E,OV)) is a package constructor
--R Abbreviation for InterfaceGroebnerPackage is INTERGB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTERGB
--R
--R-------------------------------- Operations --------------------------------
--R groebner : List(R) -> List(R)
--R
--E 1
)

)spool
)lisp (bye)

-- InterfaceGroebnerPackage.help --

InterfaceGroebnerPackage examples
Part of the Package for Algebraic Function Fields in one variable PAFF

See Also:
  o )show InterfaceGroebnerPackage
InterfaceGroebnerPackage (INTERGB)

Exports:
groebner

— package INTERGB InterfaceGroebnerPackage —

)abbrev package INTERGB InterfaceGroebnerPackage
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: April, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
InterfaceGroebnerPackage(K,symb,E,OV,R):Exports == Implementation where
  K:Field
  symb: List Symbol
  E:OrderedAbelianMonoidSup
  OV: OrderedSet
  R : PolynomialCategory(K,E,OV)

Exports ==> with
  groebner: List R -> List R

Implementation ==> add
  if ~ (K has FiniteFieldCategory) then
    GBBPackingR ==> GroebnerPackage(K,E,OV,R)
groebner(l) == groebner(l)$GBBpackR
  else
    q:PositiveInteger := (characteristic(K) pretend PositiveInteger)
CHAPTER 10. CHAPTER I

PF ==> PrimeField(q)
DPF ==> DistributedMultivariatePolynomial(symb,PF)
D ==> DistributedMultivariatePolynomial(symb,K)
JCFGBPack ==> GroebnerPackage(PF,E,OV,DPF)
GBPack ==> GroebnerPackage(K,E,OV,D)

c coerceKtoPF: K -> PF
c coerceKtoPF(a:K):PF==
  index(lookup(a)$K)$PF

c coercePFtoK: PF -> K
c coercePFtoK(a:PF):K==
  index(lookup(a)$PF)$K

c coerceRtoDwithPF: R -> DPF

c coerceRtoDwithPF(pol) ==
  map(coerceKtoPF(#1),pol)$MPolyCatFunctions2(OV,E,E,K,PF,R,DPF)

c coerceDtoRwithPF: DPF->R

c coerceDtoRwithPF(pol) ==
  map(coercePFtoK(#1),pol)$MPolyCatFunctions2(OV,E,E,PF,K,DPF,R)

c coerceRtoD: R -> D

c coerceRtoD(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,R,D)

c coerceDtoR: D->R

c coerceDtoR(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,D,R)

gs:= size()$K
if gs = q and (representationType()$K case "prime") then
groebner(l)==
  ldmp:List DPF:= [coerceRtoDwithPF(pol) for pol in l]
  gg:=groebner(ldmp)$JCFGBPack
  [coerceDtoRwithPF(pol) for pol in gg]
else
groebner(l)==
  ldmp:List D:= [coerceRtoD(pol) for pol in l]
  gg:=groebner(ldmp)$GBPack
  [coerceDtoR(pol) for pol in gg]


INTERGB.dotabb

"INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTERGB" -> "PFECAT"
Bug! Cannot precompute params and return a function which simply computes the last call. e.g. ridHack1, below.

Functions related to the binary representation of integers. These functions directly access the bits in the big integer representation and so are much faster than using a quotient loop.

--- IntegerBits.input ---

--- IntegerBits.help ---

This package provides functions to lookup bits in integers

See Also:

o )show IntegerBits
IntegerBits (INTBIT)

Exports:
  bitLength  bitCoef  bitTruth

— package INTBIT IntegerBits —

)abbrev package INTBIT IntegerBits
++ Description:
++ This package provides functions to lookup bits in integers

IntegerBits: with
  -- bitLength(n) == # of bits to represent abs(n)
  -- bitCoef (n,i) == coef of 2**i in abs(n)
  -- bitTruth(n,i) == true if coef of 2**i in abs(n) is 1

  bitLength: Integer -> Integer
  ++ bitLength(n) returns the number of bits to represent abs(n)
  bitCoef:   (Integer, Integer) -> Integer
  ++ bitCoef(n,m) returns the coefficient of 2**m in abs(n)
  bitTruth:  (Integer, Integer) -> Boolean
  ++ bitTruth(n,m) returns true if coefficient of 2**m in abs(n) is 1

  == add
  bitLength n   == INTEGER_-LENGTH(n)$Lisp
  bitCoef (n,i) == if INTEGER_-BIT(n,i)$Lisp then 1 else 0
  bitTruth(n,i) == INTEGER_-BIT(n,i)$Lisp

— INTBIT.dotabb —

"INTBIT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTBIT"]
"Package" [color="#FF4488"]
"INTBIT" -> "Package"
package COMBINAT IntegerCombinatoricFunctions

— IntegerCombinatoricFunctions.input —

)set break resume
)sys rm -f IntegerCombinatoricFunctions.output
)spool IntegerCombinatoricFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 4
)set expose add constructor OutputForm
--R
--I OutputForm is already explicitly exposed in frame frame0
--E 1

--S 2 of 4
pascalRow(n) == \[right(binomial(n,i),4) for i in 0..n\]
--R
--R Type: Void
--E 2

--S 3 of 4
displayRow(n)==output center blankSeparate pascalRow(n)
--R
--R Type: Void
--E 3

--S 4 of 4
for i in 0..7 repeat displayRow i
--R
--R Compiling function pascalRow with type NonNegativeInteger -> List(OutputForm)
--R Compiling function displayRow with type NonNegativeInteger -> Void
--R
--R Type: Void
IntegerCombinatoricFunctions package provides some standard functions in combinatorics.

The binomial(n, r) returns the number of subsets of r objects taken among n objects, i.e. \( \frac{n!}{r! \cdot (n-r)!} \)

The binomial coefficients are the coefficients of the series expansion of a power of a binomial, that is

\[
\begin{align*}
\sum_{k=0}^{n} & \binom{n}{k} x^k = (1 + x)^n
\end{align*}
\]

This leads to the famous Pascal triangle. First we expose the OutputForm domain, which is normally hidden, so we can use it to format the lines.

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:

\[
\text{pascalRow}(n) = [\text{right}(\binom{n}{i}, 4) \text{ for } i \text{ in } 0..n]
\]

and now we format the whole line so that it looks centered:

\[
\text{displayRow}(n) = \text{output center blankSeparate pascalRow}(n)
\]

and we compute the triangle

\[
\text{for } i \text{ in } 0..7 \text{ repeat displayRow } i
\]

giving the pretty result:
OutputForm
Compiling function displayRow with type NonNegativeInteger -> Void

1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
 1 5 10 10 5 1
 1 6 15 20 15 6 1
 1 7 21 35 35 21 7 1

See Also:
o )show IntegerCombinatoricFunctions
o )d op binomial
o )show OutputForm
o )help set
CHAPTER 10.  CHAPTER I

IntegerCombinatoricFunctions(I:IntegerNumberSystem): EE == II where
  Z ==> Integer
  N ==> NonNegativeInteger
  SUP ==> SparseUnivariatePolynomial
  EE ==>
  with
  binomial: (I, I) -> I
    ++ \spad{binomial(n,r)} returns the binomial coefficient
    ++ \spad{C(n,r) = n!/(r! \cdot (n-r)!)}, where \spad{n \geq r \geq 0}.
    ++ This is the number of combinations of \spad{n} objects taken \spad{r} at a time.
    ++
    ++ \spad{x [binomial(5,i) for i in 0..5]}

  factorial: I -> I
    ++ \spad{factorial(n)} returns \spad{n!}. this is the product of all
    ++ integers between 1 and \spad{n} (inclusive).
    ++ Note that \spad{0!} is defined to be 1.

  multinomial: (I, List I) -> I
    ++ \spad{multinomial(n,[m1,m2,\ldots, mk])} returns the multinomial
    ++ coefficient \spad{n!/(m1! \cdot m2! \cdot \ldots \cdot mk!)}.  

  partition: I -> I
    ++ \spad{partition(n)} returns the number of partitions of the integer \spad{n}.
    ++ This is the number of distinct ways that \spad{n} can be written as
    ++ a sum of positive integers.

  permutation: (I, I) -> I
    ++ \spad{permutation(n)} returns \spad{!P(n,r) = n!/(n-r)!}. This is
    ++ the number of permutations of \spad{n} objects taken \spad{r} at a time.

  stirling1: (I, I) -> I
    ++ \spad{stirling1(n,m)} returns the Stirling number of the first kind
    ++ denoted \spad{S[n,m]}.  

  stirling2: (I, I) -> I
    ++ \spad{stirling2(n,m)} returns the Stirling number of the second kind
    ++ denoted \spad{SS[n,m]}.  

II ==>

add
  F : Record(Fn:I, Fv:I) := [0,1]
  B : Record(Bn:I, Bm:I, Bv:I) := [0,0,0]
  S : Record(Sn:I, Sp:SUP I) := [0,0]
  P : IndexedFlexibleArray(I,0) := new(1,1)$IndexedFlexibleArray(I,0)

partition n ==
  -- This is the number of ways of expressing \spad{n} as a sum of positive
  -- integers, without regard to order. For example partition 5 = 7
  -- since 5 = 1+1+1+1+1 = 1+1+1+2 = 1+2+2 = 1+4 = 2+3 = 5 .  
  -- Uses O(\sqrt{n}) term recurrence from Abramowitz & Stegun pp. 825
  -- \spad{p(n) = \sum (-1)^j \cdot p(n-j)} where 0 < j := (3k**2-2) \mod 2 \leq n
  -- minIndex(P) ^= 0 => error "Partition: must have minIndex of 0"
  m ::= #P
  n < 0 => error "partition is not defined for negative integers"
  n < m::I => P(convert(n)@Z)
  concat_!(P, new((convert(n+1)@Z - m)::N,0)$IndexedFlexibleArray(I,0))
for i in m..convert(n)@Z repeat
  s:I := 1
  t:I := 0
  for k in 1.. repeat
    l := (3*k*k-k) quo 2
    l > i => leave
    u := l+k
    t := t + s * P(convert(i-l)@Z)
    u > i => leave
    t := t + s * P(convert(i-u)@Z)
    s := -s
  P.i := t
P(convert(n)@Z)

factorial n ==
  s,f,t : I
  n < 0 => error "factorial not defined for negative integers"
  if n <= F.Fn then s := f := 1 else (s, f) := F
  for k in convert(s+1)@Z .. convert(n)@Z by 2 repeat
    if k::I = n then t := n else t := k::I * (k+1)::I
    f := t * f
  F.Fn := n
  F.Fv := f

binomial(n, m) ==
  s,b:I
  n < 0 or m < 0 or m > n => 0
  m = 0 => 1
  n < 2*m => binomial(n, n-m)
  (s,b) := (0,1)
  if B.Bn = n then
    B.Bm = m+1 =>
      b := (B.Bv * (m+1)) quo (n-m)
      B.Bn := n
      B.Bm := m
      return(B.Bv := b)
    if m >= B.Bm then (s := B.Bm; b := B.Bv) else (s,b) := (0,1)
  for k in convert(s+1)@Z .. convert(m)@Z repeat
    b := (b*(n-k::I+1)) quo k::I
  B.Bn := n
  B.Bm := m
  B.Bv := b

multinomial(n, m) ==
  for t in m repeat t < 0 => return 0
  n < _+/m => 0
  s:I := 1
  for t in m repeat s := s * factorial t
  factorial n quo s
permutation(n, m) ==
  t:I
  m < 0 or n < m => 0
  m := n-m
  p:I := 1
  for k in convert(m+1)@Z .. convert(n)@Z by 2 repeat
    if k::I = n then t := n else t := (k*(k+1))::I
    p := p * t
  p

stirling1(n, m) ==
  -- Definition: (-1)**(n-m) S[n,m] is the number of
  -- permutations of n symbols which have m cycles.
  n < 0 or m < 1 or m > n => 0
  m = n => 1
  S.Sn = n => coefficient(S.Sp, convert(m)@Z :: N)
  x := monomial(1, 1)$SUP(I)
  S.Sn := n
  S.Sp := x
  for k in 1 .. convert(n-1)@Z repeat S.Sp := S.Sp * (x - k::SUP(I))
  coefficient(S.Sp, convert(m)@Z :: N)

stirling2(n, m) ==
  -- definition: SS[n,m] is the number of ways of partitioning
  -- a set of n elements into m non-empty subsets
  n < 0 or m < 1 or m > n => 0
  m = 1 or n = m => 1
  s:I := if odd? m then -1 else 1
  t:I := 0
  for k in 1..convert(m)@Z repeat
    s := -s
    t := t + s * binomial(m, k::I) * k::I ** (convert(n)@Z :: N)
  t quo factorial m

— COMBINAT.dotabb —
package INTFACT IntegerFactorizationPackage

--- IntegerFactorizationPackage.input ---

)set break resume
)sys rm -f IntegerFactorizationPackage.output
)spool IntegerFactorizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerFactorizationPackage
--E 1

)spool
)lisp (bye)

---

-- IntegerFactorizationPackage.help --

====================================================================
IntegerFactorizationPackage examples
====================================================================

This Package contains basic methods for integer factorization. The factor operation employs trial division up to 10,000. It then tests to see if n is a perfect power before using Pollard\'s rho method. Because Pollard\'s method may fail, the result of factor may contain composite factors. We should also employ Lenstra\'s elliptic curve method.

See Also:
o )show IntegerFactorizationPackage
IntegerFactorizationPackage (INTFACT)

Exports:
BasicMethod factor squareFree PollardSmallFactor

-- package INTFACT IntegerFactorizationPackage --

)abbrev package INTFACT IntegerFactorizationPackage
++ Description:
++ This Package contains basic methods for integer factorization.
++ The factor operation employs trial division up to 10,000. It
++ then tests to see if n is a perfect power before using Pollards
++ rho method. Because Pollards method may fail, the result
++ of factor may contain composite factors. We should also employ
++ Lenstra’s elliptic curve method.

IntegerFactorizationPackage(I): Exports == Implementation where
  I: IntegerNumberSystem

  B ==> Boolean
  FF ==> Factored I
  NNI ==> NonNegativeInteger
  LMI ==> ListMultiDictionary I
  FFE ==> Record(flg:Union(nil,"sqfr","irred","prime"),
                  fctr:I, xpnt:Integer)

Exports ==> with
  factor : I -> FF
    ++ factor(n) returns the full factorization of integer n
  squareFree : I -> FF
    ++ squareFree(n) returns the square free factorization of integer n
  BasicMethod : I -> FF
    ++ BasicMethod(n) returns the factorization
    ++ of integer n by trial division
  PollardSmallFactor: I -> Union(I,"failed")
    ++ PollardSmallFactor(n) returns a factor
    ++ of n or "failed" if no one is found
Implementation ==> add
  import IntegerRoots(I)

BasicSieve: (I, I) -> FF

squareFree

— package INTFACT IntegerFactorizationPackage —

squareFree(n:I):FF ==
  u:I
  if n<0 then (m := -n; u := -1)
  else (m := n; u := 1)
  (m > 1) and ((v := perfectSqrt m) case I) =>
    for rec in (l := factorList(sv := squareFree(v::I))) repeat
      rec.xpnt := 2 * rec.xpnt
      makeFR(u * unit sv, l)
  -- avoid using basic sieve when the lim is too big
  -- we know the sieve constants up to sqrt(100000000)
  lim := 1 + approxSqrt(m)
  lim > (100000000::I) => makeFR(u, factorList factor m)
  x := BasicSieve(m, lim)
  y :=
    ((m:= unit x) = 1) => factorList x
    (v := perfectSqrt m) case I =>
      concat_!(factorList x, ["sqfr",v,2]$FFE)
      concat_!(factorList x, ["sqfr",m,1]$FFE)
      makeFR(u, y)

PollardSmallFactor

This is Brent's optimization of Pollard's rho factoring. Brent's algorithm is about 24 percent faster than Pollard's. Pollard's algorithm has complexity $O(p^{1/2})$ where $p$ is the smallest prime factor of the composite number $N$.

Pollard's idea is based on the observation that two numbers $x$ and $y$ are congruent modulo $p$ with probability 0.5 after $1.177 \sqrt{p}$ numbers have been randomly chosen. If we try to factor $n$ and $p$ is a factor of $n$, then

$$1 < \gcd(|x - y|, n) \leq n$$
since \( p \) divides both \( |x - y| \) and \( n \).

Given a function \( f \) which generates a pseudo-random sequence of numbers we allow \( x \) to walk the sequence in order and \( y \) to walk the sequence at twice the rate. At each cycle we compute \( \gcd(|x - y|, n) \). If this GCD ever equals \( n \) then \( x = y \) which means that we have walked "all the way around the pseudo-random cycle" and we terminate with failure.

This algorithm returns failure on all primes but also fails on some composite numbers.

Quoting Brent’s back-tracking idea:

The best-known algorithm for finding GCDs is the Euclidean algorithm which takes \( O(\log N) \) times as long as one multiplication mod \( N \). Pollard showed that most of the GCD computations in Floyd’s algorithm could be dispensed with.

... The idea is simple: if \( P_F \) computes \( \gcd(z_1, N), \gcd(z_2, N), \ldots \), then we compute

\[
q_i = \prod_{j=1}^{i} z_j \pmod{N}
\]

and only compute \( \gcd(q_i, N) \) when \( i \) is a multiple of \( m \), where \( \log N << m << N^{1/4} \). Since \( q_{i+1} = q_i \times z_{i+1} \pmod{N} \), the work required for each GCD computation in algorithm \( P_F \) is effectively reduced to that for a multiplication mod \( N \) in the modified algorithm. The probability of the algorithm failing because \( q_i = 0 \) increases, so it is best not to choose \( m \) too large. This problem can be minimized by backtracking to the state after the previous GCD computation and setting \( m = 1 \).

Brent incorporates back-tracking, omits the random choice of \( u \), and makes some minor modifications. His algorithm (p192-183) reads:

\[
y := x_0; r := 1; q := 1;
\]

repeat \( x := y; \)
  for \( i := 1 \) to \( r \) do \( y := f(y); k := 0; \)
   repeat \( y_s := y; \)
     for \( i := 1 \) to \( \min(m, r - k) \) do
       begin \( y := f(y); q := q \times |x - y| \pmod{N} \)
       end;

   \( G := \gcd(q, N); k := k + m \)
   until \( (k \geq r) \) or \( (G > 1) \);

until \( G > 1 \);

if \( G = N \) then
  repeat \( y_s := f(y_s); G := \gcd(|y - yx|, N) \)
  until \( G > 1 \);

if \( G = N \) then failure else success
Here we use the function 
\[(y \ast y + 5 :: I) \text{ rem } n\]
as our pseudo-random sequence with a random starting value for y.

On possible optimization to explore is to keep a hash table for the computed values of the function \(y_{i+1} := f(y_i)\) since we effectively walk the sequence several times. And we walk the sequence in a loop many times. But because we are generating a very large number of numbers the array can be a simple array of fixed size that captures the last n values. So if we make a fixed array F of, say \(2^q\) elements we can store \(f(y_i)\) in \(F[y_i \text{ mod } 2^q]\).

One property that this algorithm assumes is that the function used to generate the numbers has a long, hopefully complete, period. It is not clear that the recommended function has that property.

```haskell
-- package INTFACT IntegerFactorizationPackage --

PollardSmallFactor(n:I):Union(I,"failed") ==
  -- Use the Brent variation
  x0 := random()$I
  m := 100 :: I
  y := x0 rem n
  r:I := 1
  q:I := 1
  G:I := 1
  until G > 1 repeat
    x := y
    for i in 1..convert(r)@Integer repeat
      y := (y\ast y + 5 :: I) \text{ rem } n
      k:I := 0
      until (k>r) or (G>1) repeat
        yss := y
        for i in 1..convert(min(m,r-k))@Integer repeat
          y := (y\ast y + 5 :: I) \text{ rem } n
          q := q*abs(x-y) \text{ rem } n
          G := gcd(q,n)
          k := k+m
        r := 2*r
      if G=n then
        until G>1 repeat
          yss := (yss\ast yss + 5 :: I) \text{ rem } n
          G := gcd(abs(x-yss),n)
        G=n => "failed"
      G
```

---
BasicSieve

We create a list of prime numbers up to the limit given. The prior code used a circular list but tests of that list show that on average more than 50 the required prime numbers. Overall this is a small percentage of the time needed to factor.

This loop uses three pieces of information

1. n which is the number we are testing
2. d which is the current prime to test
3. lim which is the upper limit of the primes to test

We loop d over the list of primes. If the remaining number n is smaller than the square of d then n must be prime and if it is not one, we add it to the list of primes. If the remaining number is larger than the square of d we remove all factors of d, reducing n each time. Then we add a record of the new factor and its multiplicity, m. We continue the loop until we run out of primes.

Annoyingly enough, primes does not return an ordered list so we fix this.

The sieve works up to a given limit, reducing out the factors that it finds. If it can find all of the factors than it returns a factored result where the first element is the unit 1. If there is still a part of the number unfactored it returns the number and a list of the factors found and their multiplicity.

Basically we just loop thru the prime factors checking to see if they are a component of the number, n. If so, we remove the factor from the number n (possibly m times) and continue thru the list of primes.

— package INTFACT IntegerFactorizationPackage —

```plaintext
BasicSieve(n, lim) ==
  p:=primes(1::I,lim::I)$IntegerPrimesPackage(I)
  l:List(I) := append([first p],reverse rest p)
  ls := empty()$List(FFE)
  for d in l repeat
    if n<d*d then
      if n>1 then ls := concat_!(ls, ["prime",n,1]$FFE)
      return makeFR(1, ls)
    for m in 0.. while zero?(n rem d) repeat n := n quo d
    if m>0 then ls := concat_!(ls, ["prime",d,convert m]$FFE)
    makeFR(n,ls)
```

BasicMethod

— package INTFACT IntegerFactorizationPackage —
The factor function is many orders of magnitude slower than the results of other systems. A posting on sci.math.symbolic showed that NTL could factor the final value (t6) in about 11 seconds. Axiom takes about 8 hours.

\begin{verbatim}
a1:=101
a2:=109
t1:=a1*a2
factor t1

a3:=21525175387
t2:=t1*a3
factor t2

a4:=218301576858349
t3:=t2*a4
factor t3

a5:=13731482973783137
t4:=t3*a5
factor t4

a6:=23326138687706820109
t5:=t4*a6
factor t5

a7:=4328240801173188438252813716944518369161
t6:=t5*a7
factor t6
\end{verbatim}

— package INTFACT IntegerFactorizationPackage —

\begin{verbatim}
factor m ==
u:I
  zero? m => 0
  if negative? m then (n := -m; u := -1)
  else (m := n; u := 1)
  x := BasicSieve(m, 1 + approxSqrt m)
  makeFR(u, factorList x)
\end{verbatim}
else (n := m; u := 1)

b := BasicSieve(n, 10000::I)
flb := factorList b
((n := unit b) = 1) => makeFR(u, flb)
a:LMI := dictionary() -- numbers yet to be factored
b:LMI := dictionary() -- prime factors found
f:LMI := dictionary() -- number which could not be factored
insert_!(n, a)
while not empty? a repeat
   n := inspect a; c := count(n, a); remove_!(n, a)
   prime?(n)$IntegerPrimesPackage(I) => insert_!(n, b, c)
   -- test for a perfect power
   (s := perfectNthRoot n).exponent > 1 =>
      insert_!(s.base, a, c * s.exponent)
   -- test for a difference of square
   x:=approxSqrt n
   if (x**2<n) then x:=x+1
   (y:=perfectSqrt (x**2-n)) case I =>
      insert_!(x+y,a,c)
      insert_!(x-y,a,c)
   (d := PollardSmallFactor n) case I =>
      for m in 0.. while zero?(n rem d) repeat n := n quo d
      insert_!(d, a, m * c)
   if n > 1 then insert_!(n, a, c)
   -- an elliptic curve factorization attempt should be made here
   insert_!(n, f, c)
   -- insert prime factors found
while not empty? b repeat
   n := inspect b; c := count(n, b); remove_!(n, b)
   flb := concat_!(flb, ["prime",n,convert c]$FFE)
   -- insert non-prime factors found
while not empty? f repeat
   n := inspect f; c := count(n, f); remove_!(n, f)
   flb := concat_!(flb, ["nil",n,convert c]$FFE)
makeFR(u, flb)

—— INTFACT.dotabb ——

"INTFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTFACT"]
"MDAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MDAGG"]
"INTFACT" -> "MDAGG"

——
package ZLINDEP IntegerLinearDependence

— IntegerLinearDependence.input —

)set break resume
)spool IntegerLinearDependence.output
)set message test on
)set message auto off
)clear all

--S 1 of 8
M := SQMATRIX(2,INT)
--R
--R
--R (1) SquareMatrix(2,Integer)
--R Type: Domain
--E 1

--S 2 of 8
m1: M := squareMatrix matrix [ [1, 2], [0, -1] ]
--R
--R
--R +1 2 +
--R (2) | |
--R +0 - 1+
--R Type: SquareMatrix(2,Integer)
--E 2

--S 3 of 8
m2: M := squareMatrix matrix [ [2, 3], [1, -2] ]
--R
--R
--R +2 3 +
--R (3) | |
--R +1 - 2+
--R Type: SquareMatrix(2,Integer)
--E 3

--S 4 of 8
m3: M := squareMatrix matrix [ [3, 4], [2, -3] ]
--R
--R
--R +3 4 +
--R (4) | |
--R +2 - 3+
--R Type: SquareMatrix(2,Integer)
--E 4
This package tests for linear dependence over the integers.

The elements \( v_1, \ldots, v_N \) of a module \( M \) over a ring \( R \) are said to be
linearly dependent over \( R \) if there exist \( c_1, \ldots, c_N \) in \( R \), not all 0, such that \( c_1 v_1 + \ldots + c_N v_N = 0 \). If such \( c_i \)'s exist, they form what is called a linear dependence relation over \( R \) for the \( v_i \)'s.

The package IntegerLinearDependence provides functions for testing whether some elements of a module over the integers are linearly dependent over the integers, and to find the linear dependence relations, if any.

Consider the domain of two by two square matrices with integer entries.

\[
M := \text{SQMATRIX}(2, \text{INT})
\]

\[
\text{SquareMatrix}(2, \text{Integer})
\]

\[
\text{Type: Domain}
\]

Now create three such matrices.

\[
m1 := \text{squareMatrix} \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}
\]

\[
\text{Type: SquareMatrix}(2, \text{Integer})
\]

\[
m2 := \text{squareMatrix} \begin{bmatrix} 2 & 3 \\ 1 & -2 \end{bmatrix}
\]

\[
\text{Type: SquareMatrix}(2, \text{Integer})
\]

\[
m3 := \text{squareMatrix} \begin{bmatrix} 3 & 4 \\ 2 & -3 \end{bmatrix}
\]

\[
\text{Type: SquareMatrix}(2, \text{Integer})
\]

This tells you whether \( m_1, m_2 \) and \( m_3 \) are linearly dependent over the integers.

\[
\text{linearlyDependentOverZ? vector } [m1, m2, m3]
\]

\[
\text{true}
\]

\[
\text{Type: Boolean}
\]

Since they are linearly dependent, you can ask for the dependence relation.

\[
c := \text{linearDependenceOverZ} \text{ vector } [m1, m2, m3] [1, -2, 1]
\]

\[
\text{Type: Union(Vector Integer, ...)}
\]

This means that the following linear combination should be 0.

\[
c.1 \times m1 + c.2 \times m2 + c.3 \times m3
\]
When a given set of elements are linearly dependent over \( \mathbb{R} \), this also means that at least one of them can be rewritten as a linear combination of the others with coefficients in the quotient field of \( \mathbb{R} \).

To express a given element in terms of other elements, use the operation `solveLinearlyOverQ`.

\[
\text{solveLinearlyOverQ(vector } [m1, m3], m2) = \begin{bmatrix} 1 & 1 \\ -1 & -1 \\ 2 & 2 \end{bmatrix}
\]

Type: Union(Vector Fraction Integer,...)

See Also:

- `)show IntegerLinearDependence`
++ Description:
++ Test for linear dependence over the integers.

IntegerLinearDependence(R): Exports == Implementation where
  R: LinearlyExplicitRingOver Integer
  Z ==> Integer

Exports == with
  linearlyDependentOverZ?: Vector R -> Boolean
  linearDependenceOverZ : Vector R -> Union(Vector Z, "failed")
  solveLinearlyOverQ : (Vector R, R) ->
    Union(Vector Fraction Z, "failed")

Implementation ==> add
  import LinearDependence(Z, R)

linearlyDependentOverZ? v == linearlyDependent? v
linearDependenceOverZ v == linearDependence v
solveLinearlyOverQ(v, c) == solveLinear(v, c)
\begin{verbatim}
)set break resume
)spool IntegerNumberTheoryFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 30
div144 := divisors(144)
--R
--R
--R  (1) [1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
--R
--E 1

--S 2 of 30
#(div144)
--R
--R
--R  (2) 15
--R
--E 2

--S 3 of 30
reduce(+,div144)
--R
--R
--R  (3) 403
--R
--E 3

--S 4 of 30
numberOfDivisors(144)
--R
--R
--R  (4) 15
--R
--E 4

--S 5 of 30
sumOfDivisors(144)
--R
--R
--R  (5) 403
--R
--E 5

--S 6 of 30
f1(n)==reduce(+,[moebiusMu(d)*numberOfDivisors(quo(n,d))_ for d in divisors(n)])
--R
\end{verbatim}
--R Type: Void
--E 6

--S 7 of 30
f1(200)
--R
--R Compiling function f1 with type PositiveInteger -> Integer
--R
--R (7) 1
--R Type: PositiveInteger
--E 7

--S 8 of 30
f1(846)
--R
--R
--R (8) 1
--R Type: PositiveInteger
--E 8

--S 9 of 30
f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d)) for d in divisors(n)])
--R
--R Type: Void
--E 9

--S 10 of 30
f2(200)
--R
--R Compiling function f2 with type PositiveInteger -> Integer
--R
--R (10) 200
--R Type: PositiveInteger
--E 10

--S 11 of 30
f2(846)
--R
--R
--R (11) 846
--R Type: PositiveInteger
--E 11

--S 12 of 30
fibonacci(25)
--R
--R
--R (12) 75025
--R Type: PositiveInteger
[fibonacci(n) for n in 1..15]

\[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610\]

Type: List(Integer)

\(\text{fib}(n) == \text{reduce}(+, [\binomial(n-1-k,k) \text{ for } k \text{ in } 0..\text{quo}(n-1,2)])\)

Type: Void

\(\text{fib}(25)\)

\(75025\)

Type: PositiveInteger

\(\text{legendre}(3,5)\)

\(-1\)

Type: Integer

\(\text{legendre}(23,691)\)

\(-1\)

Type: Integer
h(d) == quo(reduce(+,[jacobi(d,k) for k in 1..quo(-d, 2)]),2-jacobi(d,2))
--R
--R Type: Void
--E 19

--S 20 of 30
h(-163)
--R
--R Compiling function h with type Integer -> Integer
--R
--R (20) 1
--R Type: PositiveInteger
--E 20

--S 21 of 30
h(-499)
--R
--R
--R (21) 3
--R Type: PositiveInteger
--E 21

--S 22 of 30
h(-1832)
--R
--R
--R (22) 26
--R Type: PositiveInteger
--E 22

--S 23 of 30
inverse:(INT,INT)->INT
--R
--R Type: Void
--E 23

--S 24 of 30
inverse(a,b) ==
  borg:INT:=b
  c1:INT := 1
  d1:INT := 0
  while b ~= 0 repeat
    q := a quo b
    r := a-q*b
    print [a, "=", q, "*", b, "+", r]
    (a,b):=(b,r)
    (c1,d1):=(d1,c1-q*d1)
  a ~= 1 => error("moduli are not relatively prime")
  positiveRemainder(c1,borg)
--R
CHAPTER 10.  CHAPTER I

--R
--E 24

--S 25 of 30
inverse(15,26)
--R
--R Compiling function inverse with type (Integer,Integer) -> Integer
--R [15,"=","0","*","(",26,"),"+=",15]
--R [26,"=",1,"*","(",15,"),"+=",11]
--R [4,"=",1,"*","(",3,"),"+=",1]
--R [3,"=",3,"*","(",1,"),"+=",0]
--R
--R (25) 7
--R
--E 25

--S 26 of 30
x1:=4
--R
--R
--R (26) 4
--R
--E 26

--S 27 of 30
m1:=5
--R
--R
--R (27) 5
--R
--E 27

--S 28 of 30
x2:=2
--R
--R
--R (28) 2
--R
--E 28

--S 29 of 30
m2:=3
--R
--R
--R (29) 3
--R
--E 29
This package provides various number theoretic functions on the integers.

The IntegerNumberTheoryFunctions package contains a variety of operations of interest to number theorists. Many of these operations deal with divisibility properties of integers. (Recall that an integer a divides an integer b if there is an integer c such that b = a * c.)

The operation divisors returns a list of the divisors of an integer.

\[
\text{div}144 := \text{divisors}(144) \\
[1,2,3,4,6,8,9,12,16,18,24,36,48,72,144] \\
\text{Type: List Integer}
\]

You can now compute the number of divisors of 144 and the sum of the divisors of 144 by counting and summing the elements of the list we just created.

\[
\#(\text{div}144) \\
15 \\
\text{Type: PositiveInteger}
\]

\[
\text{reduce}(+,\text{div}144) \\
403 \\
\text{Type: PositiveInteger}
\]

Of course, you can compute the number of divisors of an integer n, usually denoted \(d(n)\), and the sum of the divisors of an integer n, usually denoted \(\sigma(n)\), without ever listing the divisors of n.

In Axiom, you can simply call the operations \text{numberOfDivisors} and \text{sigma}.
CHAPTER 10. CHAPTER I

sumOfDivisors.

numberOfDivisors(144)
15
Type: PositiveInteger

sumOfDivisors(144)
403
Type: PositiveInteger

The key is that d(n) and sigma(n) are "multiplicative functions.
This means that when n and m are relatively prime, that is, when
n and m have no prime factor in common, then d(nm) = d(n) d(m) and
sigma(nm) = sigma(n) sigma(m). Note that these functions are trivial to
calculate when n is a prime power and are computed for general n from the
prime factorization of n. Other examples of multiplicative functions
are sigma_k(n), the sum of the k-th powers of the divisors of n and
varphi(n), the number of integers between 1 and n which are prime to n.
The corresponding Axiom operations are called sumOfKthPowerDivisors and
eulerPhi.

An interesting function is mu(n), the Moebius mu function, defined as
follows: mu(1) = 1, mu(n) = 0, when n is divisible by a square, and
mu = (-1)^k, when n is the product of k distinct primes. The corresponding
Axiom operation is moebiusMu. This function occurs in the following theorem:

Theorem: (Moebius Inversion Formula):
Let f(n) be a function on the positive integers and let F(n)
be defined by
F(n) = \sum_{d | n} f(n)
the sum of f(n) over d | n where the sum is taken over the
positive divisors of n. Then the values of f(n) can be recovered
from the values of F(n): f(n) = \sum_{d | n} \mu(n) F(n/d) where
again the sum is taken over the positive divisors of n.

When f(n) = 1, then F(n) = d(n). Thus, if you sum mu(d).d(n/d) over
the positive divisors d of n, you should always get 1.

f1(n)==reduce(+,[moebiusMu(d)*numberOfDivisors(quo(n,d)) for d in divisors(n)])
Type: Void

f1(200)
1
Type: PositiveInteger

f1(846)
1
Type: PositiveInteger
Similarly, when \( f(n) = n \), then \( F(n) = \sigma(n) \). Thus, if you sum
\( \mu(d) \cdot \sigma(n/d) \) over the positive divisors \( d \) of \( n \), you should always get \( n \).

\[
f_2(n) = \text{reduce}(+, [\mu(d) \cdot \sum \text{divisors}(\text{quo}(n,d))]
\text{for } d \text{ in } \text{divisors}(n))
\]

Type: Void

\[
f_2(200) = 200
\]

Type: PositiveInteger

\[
f_2(846) = 846
\]

Type: PositiveInteger

The Fibonacci numbers are defined by
\( F(1) = F(2) = 1 \) and
\( F(n) = F(n-1) + F(n-2) \) for \( n = 3,4,... \).

The operation fibonacci computes the \( n \)-th Fibonacci number.

\[
fibonacci(25) = 75025
\]

Type: PositiveInteger

\[
[fibonacci(n) \text{ for } n \text{ in } 1..15]
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
\]

Type: List Integer

Fibonacci numbers can also be expressed as sums of binomial coefficients.

\[
fib(n) = \text{reduce}(+, [\binomial(n-1-k,k) \text{ for } k \text{ in } 0..\text{quo}(n-1,2)])
\]

Type: Void

\[
fib(25) = 75025
\]

Type: PositiveInteger

\[
[fib(n) \text{ for } n \text{ in } 1..15]
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
\]

Type: List Integer

Quadratic symbols can be computed with the operations legendre and
jacobi. The Legendre symbol \( a/p \) is defined for integers \( a \) and \( p \) with
\( p \) an odd prime number. By definition,

\[
(a/p) = +1, \text{ when } a \text{ is a square } (\text{mod } p),
(a/p) = -1, \text{ when } a \text{ is not a square } (\text{mod } p), \text{ and}
(a/p) = 0, \text{ when } a \text{ is divisible by } p.
\]
You compute \((a/p)\) via the command `legendre(a,p)`.

\[
\begin{align*}
\text{legendre}(3,5) & \quad -1 \quad \text{Type: Integer} \\
\text{legendre}(23,691) & \quad -1 \quad \text{Type: Integer}
\end{align*}
\]

The Jacobi symbol \((a/n)\) is the usual extension of the Legendre symbol, where \(n\) is an arbitrary integer. The most important property of the Jacobi symbol is the following: if \(K\) is a quadratic field with discriminant \(d\) and quadratic character \(\chi\), then \(\chi(n) = (d/n)\). Thus, you can use the Jacobi symbol to compute, say, the class numbers of imaginary quadratic fields from a standard class number formula.

This function computes the class number of the imaginary quadratic field with discriminant \(d\).

\[
\begin{align*}
\text{h}(d) & \quad = \quad \text{quo}(\text{reduce}(+, [\text{jacobi}(d,k) \text{ for } k \text{ in } 1..\text{quo}(-d, 2)])), 2\text{-jacobi}(d,2)) \\
& \quad \text{Type: Void}
\end{align*}
\]

\[
\begin{align*}
\text{h}(-163) & \quad 1 \quad \text{Type: PositiveInteger} \\
\text{h}(-499) & \quad 3 \quad \text{Type: PositiveInteger} \\
\text{h}(-1832) & \quad 26 \quad \text{Type: PositiveInteger}
\end{align*}
\]

====================================================================

The inverse function

====================================================================

The inverse function is derived from the Extended Euclidean Algorithm. If we divide one integer by another nonzero integer we get an integer quotient plus a remainder which is, in general, a rational number. For instance,

\[
13/5 = 2 + 3/5
\]

where 2 is the quotient and 3/5 is the remainder.

If we multiply thru by the denominator of the remainder we get an answer in integer terms which no longer involves division:

\[
13 = 2(5) + 3
\]
This gives a method of dividing integers. Specifically, if \( a \) and \( b \) are positive integers, there exist unique non-negative integers \( q \) and \( r \) so that
\[
a = qb + r, \quad \text{where } 0 \leq r < b
\]
\( q \) is called the quotient and \( r \) the remainder.

The greatest common divisor of integers \( a \) and \( b \), denoted by \( \gcd(a, b) \), is the largest integer that divides (without remainder) both \( a \) and \( b \). So, for example:
\[
\gcd(15, 5) = 5, \\
\gcd(7, 9) = 1, \\
\gcd(12, 9) = 3, \\
\gcd(81, 57) = 3.
\]

The \( \gcd \) of two integers can be found by repeated application of the division algorithm, this is known as the Euclidean Algorithm. You repeatedly divide the divisor by the remainder until the remainder is 0. The \( \gcd \) is the last non-zero remainder in this algorithm. The following example shows the algorithm.

Finding the \( \gcd \) of 81 and 57 by the Euclidean Algorithm:
\[
81 = 1(57) + 24 \\
57 = 2(24) + 9 \\
24 = 2(9) + 6 \\
9 = 1(6) + 3 \\
6 = 2(3) + 0
\]
So the greatest common divisor, the \( \text{GCD}(81,51) = 3 \).

If the \( \gcd(a, b) = r \) then there exist integers \( s \) and \( t \) so that
\[
s(a) + t(b) = r
\]
By back substitution in the steps in the Euclidean Algorithm, it is possible to find these integers \( s \) and \( t \). We shall do this with the above example:

Starting with the next to last line, we have:
\[
3 = 9 - 1(6)
\]
From the line before that, we see that 6 = 24 - 2(9), so:
\[
3 = 9 - 1(24 - 2(9)) = 3(9) - 1(24)
\]
From the line before that, we have 9 = 57 - 2(24), so:
\[
3 = 3( 57 - 2(24)) - 1(24) = 3(57) - 7(24)
\]
And, from the line before that $24 = 81 - 1(57)$, giving us:

$$3 = 3(57) - 7(81 - 1(57)) = 10(57) - 7(81)$$

So we have found $s = -7$ and $t = 10$.

The Extended Euclidean Algorithm computes the GCD(a,b) and the values for $s$ and $t$.

Suppose we were doing arithmetics modulo 26 and we needed to find the inverse of a number mod 26. This turned out to be a difficult task (and not always possible). We observed that a number $x$ had an inverse mod 26 (i.e., a number $y$ so that $xy = 1 \mod 26$) if and only if $\gcd(x, 26) = 1$. In the general case the inverse of $x$ exists if and only if $\gcd(x, n) = 1$ and if it exists then there exist integers $s$ and $t$ so that

$$sx + tn = 1$$

But this says that $sx = 1 + (-t)n$, or in other words,

$$sx \equiv 1 \mod n$$

So, $s$ (reduced mod $n$ if need be) is the inverse of $x$ mod $n$.

The extended Euclidean algorithm calculates $s$ efficiently.

```
Finding the inverse mod n
```

We will number the steps of the Euclidean algorithm starting with step 0. The quotient obtained at step $i$ will be denoted by $q_i$ and an auxiliary number, $s_i$. For the first two steps, the value of this number is given:

- $s(0) = 0$
- $s(1) = 1$

For the remainder of the steps, we recursively calculate

$$s(i) = s(i-2) - s(i-1) q(i-2) \mod n$$

The algorithm starts by "dividing" $n$ by $x$. If the last non-zero remainder occurs at step $k$, then if this remainder is 1, $x$ has an inverse and it is $s(k+2)$. If the remainder is not 1, then $x$ does not have an inverse.

For example, find the inverse of 15 mod 26.

```
Step 0: 26 = 1(15) + 11 $s(0) = 0$
Step 1: 15 = 1(11) + 4 $s(1) = 1$
Step 2: 11 = 2(4) + 3 $s(2) = 0 - 1(1) \mod 26 = 25$
Step 3: 4 = 1(3) + 1 $s(3) = 1 - 25(1) \mod 26 = -24 \mod 26 = 2$
Step 4: 3 = 3(1) + 0 $s(4) = 25 - 2(2) \mod 26 = 21$
```

$s(5) = 2 - 21(1) \mod 26 = -19 \mod 26 = 7$
Notice that $15(7) = 105 = 1 + 4(26) = 1 \pmod{26}$.

Using the half extended Euclidean algorithm we compute $1/a \mod b$.

```lisp
inverse: (INT, INT) -> INT
Type: Void
inverse(a, b) ==
borg: INT := b
cl: INT := 1
d1: INT := 0
while b ~= 0 repeat
  q := a quo b
  r := a-q*b
  print [a, "=", q, "*", b, "+", r]
  (a, b) := (b, r)
  (c1, d1) := (d1, c1-q*d1)
a ~= 1 => error("moduli are not relatively prime")
positiveRemainder(c1, borg)
Type: Void
```

```lisp
inverse(15, 26)
[15,"=",0,"*",26,"+",15]
[4,"=",1,"*",3,"+",1]
[3,"=",3,"*",1,"+",0]

7
Type: PositiveInteger
```

The Chinese Remainder Algorithm

The Chinese Remainder Theorem says that given $n$ moduli $m_i$ for $(1 \leq i \leq n)$ of pairwise coprime integers and a set of congruential equations $x \equiv c_i \pmod{m_i}$ $(1 \leq i \leq n)$ for an arbitrary integer $x$, there is a unique solution $c$ for $x \mod M$, say $x \equiv c \pmod{M}$.

If the upper bound $B$ of the absolute value of $x$ is known, that is, if $|x| \leq B$ then the number of moduli can be chosen such that $B < M/2$. If $c$ is the absolutely least residue modulo $M$ then $c$ and $x$ coincide so $x$ is uniquely determined.

Let $m_1, m_2, \ldots, m_r$ be positive integers that are pairwise relatively prime.
Let $x_1, x_2, \ldots, x_r$ be integers with $0 \leq x_i < m_i$. Then, there is exactly one $x$ in the interval $0 \leq x < m_1 \ldots m_2 \ldots m_r$ that satisfies the remainder equations

$$\text{irem}(x, m_i) = x_i, \ i = 1, 2, \ldots, r$$

where $\text{irem}$ is the positive integer remainder function.

For example, let $x_1 = 4$, $m_1 = 5$, $x_2 = 2$, $m_2 = 3$. We know that

$$\text{irem}(x, m_1) = x_1$$
$$\text{irem}(x, m_2) = x_2$$

where $0 \leq x_1 < m_1$ and $0 \leq x_2 < m_2$.

By the extended Euclidean Algorithm there are integers $c$ and $d$ such that

$$c \cdot m_1 + d \cdot m_2 = 1$$

In this case we are looking for an integer such that

$$\text{irem}(x, 5) = 4,$$
$$\text{irem}(x, 3) = 2$$

The algorithm we use is to first compute the positive integer remainder of $x_1$ and $m_1$ to get a new $x_1$:

$$x_1 = \text{positiveRemainder}(x_1, m_1)$$
$$4 = \text{positiveRemainder}(4, 5)$$

Next compute the positive integer remainder of $x_2$ and $m_2$ to get a new $x_2$:

$$x_2 = \text{positiveRemainder}(x_2, m_2)$$
$$2 = \text{positiveRemainder}(2, 3)$$

Then we compute $x_1 + m_1 \ldots \text{positiveRemainder}(((x_2-x_1)\cdot\text{inverse}(m_1,m_2)), m_2)$ or

$$4 + 5 \cdot \text{positiveRemainder}(((2-4)\cdot\text{inverse}(5,3)), 3)$$

or

$$4 + 5 \cdot \text{positiveRemainder}(-2\cdot2), 3)$$

or

$$4 + 5 \cdot 2$$

or

$$14$$

This function has a restricted signature which only allows for computing the Chinese remainder of two numbers and two moduli.

$x_1 := 4$

Type: PositiveInteger

$m_1 := 5$

Type: PositiveInteger

$x_2 := 2$
2

Type: PositiveInteger

m2:=3

Type: PositiveInteger

result:=chineseRemainder(x1,m1,x2,m2)

14

Type: PositiveInteger

See Also:

::show IntegerNumberTheoryFunctions

IntegerNumberTheoryFunctions (INTHEORY)

Exports:
+ bernoulli +
+ chineseRemainder +
+ divisors +
+ euler +
+ eulerPhi +
+ fibonacci +
+ harmonic +
+ jacobi +
+ legendre +
+ moebiusMu +
+ numberOfDivisors +
+ sumOfDivisors +
+ sumOfKthPowerDivisors +

package INTHEORY IntegerNumberTheoryFunctions —

)abbrev package INTHEORY IntegerNumberTheoryFunctions
++ Author: Michael Monagan, Martin Brock, Robert Sutor, Timothy Daly
++ Date Created: June 1987
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various number theoretic functions on the integers.

IntegerNumberTheoryFunctions(): Exports == Implementation where
+ I ==> Integer +
+ RN ==> Fraction I +
+ SUP ==> SparseUnivariatePolynomial +
\textbf{NNI} \Rightarrow \textbf{NonNegativeInteger}

\textbf{Exports} \Rightarrow \textbf{with}

bernoulli : I \rightarrow \text{RN}
++ \texttt{bernoulli(n)} returns the nth Bernoulli number.
++ this is \texttt{B(n,0)}, where \texttt{B(n,x)} is the \texttt{n}th Bernoulli
++ polynomial.

\textbf{chineseRemainder:} (I,I,I,I) \rightarrow I
++ \texttt{chineseRemainder(x1,m1,x2,m2)} returns \(w\), where \(w\) is such that
++ \texttt{w = x1 \mod m1} and \texttt{w = x2 \mod m2}. Note that \texttt{m1} and
++ \texttt{m2} must be relatively prime.

divisors : I \rightarrow \text{List I}
++ \texttt{divisors(divisors(n))} returns a list of the divisors of \(n\).

euler : I \rightarrow I
++ \texttt{euler(n)} returns the \(n\)th Euler number. This is
++ \texttt{2^n E(n,1/2)}, where \texttt{E(n,x)} is the \(n\)th Euler polynomial.

eulerPhi : I \rightarrow I
++ \texttt{eulerPhi(n)} returns the number of integers between 1 and \(n\)
++ (including 1) which are relatively prime to \(n\). This is the Euler phi
++ function \texttt{\phi(n)} is also called the totient function.

fibonacci : I \rightarrow I
++ \texttt{fibonacci(n)} returns the \(n\)th Fibonacci number. The Fibonacci
++ numbers \texttt{F[n]} are defined by \texttt{F[0] = F[1] = 1} and
++ \texttt{F[n] = F[n-1] + F[n-2]}.
++ The algorithm has running time \texttt{O(log(n)^3)}.
++ Reference: Knuth, The Art of Computer Programming

harmonic : I \rightarrow \text{RN}
++ \texttt{harmonic(n)} returns the nth harmonic number. This is
++ \texttt{H[n] = sum(1/k, k=1..n)}.

jacobi : (I,I) \rightarrow I
++ \texttt{Jacobi(a,b)} returns the Jacobi symbol \texttt{J(a/b)}.
++ When \(b\) is odd, \texttt{J(a/b) = product(L(a/p) for p in factor b)}.
++ Note that by convention, 0 is returned if \texttt{gcd(a,b) = 1}.
++ Iterative \texttt{O(log(b)^2)} version coded by Michael Monagan June 1987.

legendre : (I,I) \rightarrow I
++ \texttt{legendre(a,p)} returns the Legendre symbol \texttt{L(a/p)}.
++ \texttt{L(a/p) = (-1)**((p-1)/2) mod p} (\(p\) prime), which is 0 if \texttt{a}
++ is 0, 1 if \texttt{a} is a quadratic residue \texttt{mod p} and -1 otherwise.
++ Note that because the primality test is expensive,
++ if it is known that \(p\) is prime then use \texttt{Jacobi(a,p)}.

moebiusMu : I \rightarrow I
++ \texttt{mu(n)} returns the Moebius function \texttt{\mu(n)}.
++ \texttt{\mu(n)} is either \(-1, 0\) or \(1\) as follows:
++ \texttt{\mu(n) = 0} if \(n\) is divisible by a square > 1,
++ \texttt{\mu(n) = (-1)^k} if \(n\) is square-free and has \(k\) distinct
++ prime divisors.

numberOfDivisors : I \rightarrow I
++ \texttt{numberOfDivisors(n)} returns the number of integers between 1 and \(n\)
++ (inclusive) which divide \(n\). The number of divisors of \(n\) is often
++ denoted by \spad{\tau(n)}.

\hfill sumOfDivisors : I \to I
++ \spad{\text{sumOfDivisors(n)}} returns the sum of the integers between 1 and n
++ (inclusive) which divide n. The sum of the divisors of n is often
++ denoted by \spad{\sigma(n)}.

\hfill sumOfKthPowerDivisors : (I,NNI) \to I
++ \spad{\text{sumOfKthPowerDivisors(n,k)}} returns the sum of the \spad{k}\text{th}
++ powers of the integers between 1 and n (inclusive) which divide n.
++ the sum of the \spad{k}\text{th} powers of the divisors of n is often denoted
++ by \spad{\sigma_k(n)}.

Implementation \Rightarrow add
import IntegerPrimesPackage(I)

-- we store the euler and bernoulli numbers computed so far in
-- a Vector because they are computed from an n-term recurrence
E: IndexedFlexibleArray(I,0) := new(1,1)
B: IndexedFlexibleArray(RN,0) := new(1,1)
H: Record(Hn:I, Hv:RN) := [1,1]

harmonic n ==

s:I; h:RN
n < 0 \Rightarrow \text{error("harmonic not defined for negative integers")}
if n >= H.Hn then (s, h) := H else (s := 0; h := 0)
for k in s+1..n repeat h := h + 1/k
H.Hn := n
H.Hv := h
h

fibonacci n ==

n = 0 \Rightarrow 0
n < 0 \Rightarrow (\text{odd? n} \Rightarrow 1; -1) \ast \text{fibonacci(-n)}
f1, f2 : I
(f1, f2) := (0,1)
for k in length(n)-2 .. 0 by -1 repeat
  t := f2**2
  (f1, f2) := (t+f1**2, t+2*f1*f2)
  if \text{bit?(n,k)} then (f1, f2) := (f2, f1+f2)
f2

euler n ==

n < 0 \Rightarrow \text{error "euler not defined for negative integers"}
odd? n \Rightarrow 0
l := (#E) :: I
n < 1 \Rightarrow E(n)
concat!(E, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(I,0))
for i in 1 .. 1 by 2 repeat E(i) := 0
-- compute E(i) i = 1+2, 1+4, ..., n given E(j) j = 0, 2, ..., i-2
t, e : I
for i in l+1 .. n by 2 repeat
  t := e := 1
for \( j \) in 2 .. \( i-2 \) by 2 repeat
\[
\begin{align*}
t &:= (t*(i-j+1)*(i-j+2)) \text{ quo } (j*(j-1)) \\
e &:= e + t*E(j) \\
E(i) &:= -e
\end{align*}
\]
\( E(n) \)

\texttt{bernoulli \( n \) ==}
\texttt{n < 0 \Rightarrow \text{error "bernoulli not defined for negative integers"}}
\texttt{odd? \( n \) \Rightarrow}
\texttt{n = 1 \Rightarrow -1/2}
\texttt{0}
\texttt{l := (#B) :: I}
\texttt{n < l \Rightarrow B(n)}
\texttt{concat_! (B, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(RN,0))}
\texttt{for i in 1 .. \( l \) by 2 repeat B(i) := 0}
\texttt{-- compute B(i) i = l+2,l+4,...,n given B(j) j = 0,2,...,i-2}
\texttt{for i in l+1 .. \( n \) by 2 repeat}
\texttt{t:I := 1}
\texttt{b := (1-i)/2}
\texttt{for j in 2 .. \( i-2 \) by 2 repeat}
\texttt{t := (t*(i-j+2)*(i-j+3)) \text{ quo } (j*(j-1))}
\texttt{b := b + (t::RN) * B(j)}
\texttt{B(i) := -b/((i+1)::RN)}
\texttt{B(n)}

\texttt{inverse : (I,I) -> I}

\texttt{inverse(a,b) ==}
\texttt{borg:I:=b}
\texttt{cl1:I := 1}
\texttt{dl1:I := 0}
\texttt{while b ^= 0 repeat}
\texttt{q:I := a quo b}
\texttt{r:I := a-q*b}
\texttt{\langle a\rangle,\langle b\rangle:=(b,r)}
\texttt{\langle c1,d1\rangle:=(d1,c1-q*d1)}
\texttt{a ^= 1 => error("moduli are not relatively prime")}
\texttt{positiveRemainder(c1,borg)}

\texttt{chineseRemainder(x1,m1,x2,m2) ==}
\texttt{m1 < 0 or m2 < 0 \Rightarrow \text{error "moduli must be positive"}}
\texttt{x1 := positiveRemainder(x1,m1)}
\texttt{x2 := positiveRemainder(x2,m2)}
\texttt{x1 + m1 * positiveRemainder((x2-x1) * inverse(m1,m2)),m2)}

\texttt{jacobi(a,b) ==}
\texttt{-- Revised by Clifton Williamson January 1989.}
\texttt{-- Previous version returned incorrect answers when \( b \) was even.}
\texttt{-- The formula \( J(a/b) = \prod ( L(a/p) \text{ for } p \text{ in factor } b) \) is only}
\texttt{-- valid when \( b \) is odd (the Legendre symbol \( L(a/p) \) is not defined}
-- for p = 2). When b is even, the Jacobi symbol J(a/b) is only
-- defined for a = 0 or 1 (mod 4). When a = 1 (mod 8),
-- J(a/2) = +1 and when a = 5 (mod 8), we define J(a/2) = -1.
-- Extending by multiplicativity, we have J(a/b) for even b and
-- appropriate a.
-- We also define J(a/1) = 1.
-- The point of this is the following: if d is the discriminant of
-- a quadratic field K and chi is the quadratic character for K,
-- then J(d/n) = chi(n) for n > 0.
-- Reference: Hecke, Vorlesungen ueber die Theorie der Algebraischen
-- Zahlen.
if b < 0 then b := -b
b = 0 => error "second argument of jacobi may not be 0"
b = 1 => 1
even? b and positiveRemainder(a,4) > 1 =>
  error "J(a/b) not defined for b even and a = 2 or 3 (mod 4)"
even? b and even? a => 0
for k in 0.. while even? b repeat b := b quo 2
j: I := (odd? k and positiveRemainder(a,8) = 5 => -1; 1)
b = 1 => j
a := positiveRemainder(a,b)
-- assertion: 0 < a < b and odd? b
while a > 1 repeat
  if odd? a then
    -- J(a/b) = J(b/a) (-1) ** (a-1)/2 (b-1)/2
    if a rem 4 = 3 and b rem 4 = 3 then j := -j
    (a,b) := (b rem a,a)
  else
    -- J(2*a/b) = J(a/b) (-1) (b**2-1)/8
    for k in 0.. until odd? a repeat a := a quo 2
    if odd? k and (b+2) rem 8 > 4 then j := -j
    a = 0 => 0
  j
legendre(a,p) ==
  prime? p => jacobi(a,p)
  error "characteristic of legendre must be prime"
eulerPhi n ==
  n = 0 => 0
  r : RN := 1
  for entry in factors factor n repeat
    r := ((entry.factor - 1) /$RN entry.factor) * r
  numer(n * r)
divisors n ==
  oldList : List Integer := [1]
  for f in factors factor n repeat
    newList : List Integer := oldList
    for k in 1..f.exponent repeat
pow := f.factor ** k
for m in oldList repeat
    newList := concat(pow * m, newList)
oldList := newList
sort((i1:Integer, i2:Integer):Boolean +-> i1 < i2, oldList)

numberOfDivisors n ==
n = 0 => 0
*/[1+entry.exponent for entry in factors factor n]

sumOfDivisors n ==
n = 0 => 0
r : RN := */[(entry.factor**(entry.exponent::NNI + 1)-1)/
    (entry.factor-1) for entry in factors factor n]
numer r

sumOfKthPowerDivisors(n, k) ==
n = 0 => 0
r : RN := */[(entry.factor**(k*entry.exponent::NNI+k)-1)/
    (entry.factor**k-1) for entry in factors factor n]
numer r

moebiusMu n ==
n = 1 => 1
t := factor n
for k in factors t repeat
    k.exponent > 1 => return 0
od
odd? numberOfFactors t => -1
1

package PRIMES IntegerPrimesPackage

We've expanded the list of small primes to include those between 1 and 10000.

— IntegerPrimesPackage.input —

)set break resume
The IntegerPrimesPackage implements a modification of Rabin’s probabilistic primality test and the utility functions nextPrime, prevPrime, and primes.

See Also:
o )show IntegerPrimesPackage

Exports:
nextPrime prevPrime prime? primes
--- package PRIMES IntegerPrimesPackage ---

)abbrev package PRIMES IntegerPrimesPackage
++ Author: Michael Monagan, James Davenport
++ Date Created: August 1987
++ Date Last Updated: 31 May 1993
++ References:
++ J. H. Davenport ‘‘Primality testing revisited’’ Technical Report TR2/93
++ (ATR/6) (NP2556) Numerical Algorithms Group, Inc., Downer’s Grove, IL, USA
++ and Oxford, UK, August 1993
++ Description:
++ The \spadtype{IntegerPrimesPackage} implements a modification of
++ Rabin’s probabilistic
++ primality test and the utility functions \spadfun{nextPrime},
++ \spadfun{prevPrime} and \spadfun{primes}.

IntegerPrimesPackage(I:IntegerNumberSystem): with
  prime?: I -> Boolean
  nextPrime: I -> I
  prevPrime: I -> I
  primes: (I,I) -> List I

  ++ \spad{prime?(n)} returns true if n is prime and false if not.
  ++ The algorithm used is Rabin’s probabilistic primality test
  ++ If \spad{prime? n} returns false, n is proven composite.
  ++ If \spad{prime? n} returns true, prime? may be in error
  ++ however, the probability of error is very low.
  ++ and is zero below 25*10**9 (due to a result of Pomerance et al),
  ++ below 10**12 and 10**13 due to results of Pinch,
  ++ and below 341550071728321 due to a result of Jaeschke.
  ++ Specifically, this implementation does at least 10 pseudo prime
  ++ tests and so the probability of error is \spad{< 4**(-10)}.
  ++ The running time of this method is cubic in the length
  ++ of the input n, that is \spad{O((log n)**3)}, for n<10**20.
  ++ beyond that, the algorithm is quartic, \spad{O((log n)**4)}.
  ++ Two improvements due to Davenport have been incorporated
  ++ which catches some trivial strong pseudo-primes, such as
  ++ [Jaeschke, 1991] 1377161253229053 * 413148375987157, which
  ++ the original algorithm regards as prime
  nextPrime: I -> I
  ++ \spad{nextPrime(n)} returns the smallest prime strictly larger than n
  prevPrime: I -> I
  ++ \spad{prevPrime(n)} returns the largest prime strictly smaller than n
  primes: (I,I) -> List I
  ++ \spad{primes(a,b)} returns a list of all primes p with
  ++ \spad{a <= p <= b}
smallPrimes

This is a table of all of the primes in \([2..10000]\). It is used by the prime? function to check
for primality. It is used by the primes function to generate arrays of primes in a given range.
Changing the range included in this table implies changing the value of the nextSmallPrime
variable. There is a constant in the function squareFree from IntegerFactorizationPackage
that is the square of the upper bound of the table range, in this case 10000000.

| package PRIMES IntegerPrimesPackage |
| smallPrimes: List I :=
2::I, 3::I, 5::I, 7::I, 11::I, 13::I, 17::I, 19::I, _
23::I, 29::I, 31::I, 37::I, 41::I, 43::I, 47::I, 53::I, _
59::I, 61::I, 67::I, 71::I, 73::I, 79::I, 83::I, 89::I, _
97::I, 101::I, 103::I, 107::I, 109::I, 113::I, 127::I, _
131::I, 137::I, 139::I, 149::I, 151::I, 157::I, 163::I, _
167::I, 173::I, 179::I, 181::I, 191::I, 193::I, 197::I, _
199::I, 211::I, 223::I, 227::I, 229::I, 233::I, 239::I, _
241::I, 251::I, 257::I, 263::I, 269::I, 271::I, 277::I, _
281::I, 283::I, 293::I, 307::I, 311::I, 313::I, 317::I, _
331::I, 337::I, 347::I, 349::I, 353::I, 359::I, 367::I, _
373::I, 379::I, 383::I, 389::I, 397::I, 401::I, 409::I, _
419::I, 421::I, 431::I, 433::I, 439::I, 443::I, 449::I, _
457::I, 461::I, 463::I, 467::I, 479::I, 487::I, 491::I, _
499::I, 503::I, 509::I, 521::I, 523::I, 541::I, 547::I, _
557::I, 563::I, 569::I, 571::I, 577::I, 587::I, 593::I, _
599::I, 601::I, 607::I, 613::I, 617::I, 619::I, 631::I, _
641::I, 643::I, 647::I, 653::I, 659::I, 661::I, 673::I, _
677::I, 683::I, 691::I, 701::I, 709::I, 719::I, 727::I, _
733::I, 739::I, 743::I, 751::I, 757::I, 761::I, 769::I, _
773::I, 787::I, 797::I, 809::I, 811::I, 821::I, 823::I, _
827::I, 829::I, 839::I, 853::I, 857::I, 859::I, 863::I, _
877::I, 881::I, 883::I, 887::I, 907::I, 911::I, 919::I, _
929::I, 937::I, 941::I, 947::I, 953::I, 967::I, 971::I, _
977::I, 983::I, 991::I, 997::I, 1009::I, 1013::I, _
1019::I, 1021::I, 1031::I, 1033::I, 1039::I, 1049::I, _
1051::I, 1061::I, 1063::I, 1069::I, 1087::I, 1091::I, _
1093::I, 1097::I, 1103::I, 1109::I, 1117::I, 1123::I, _
1129::I, 1151::I, 1153::I, 1163::I, 1171::I, 1181::I, _
1187::I, 1193::I, 1201::I, 1213::I, 1217::I, 1223::I, _
1229::I, 1231::I, 1237::I, 1249::I, 1259::I, 1277::I, _
1279::I, 1283::I, 1289::I, 1291::I, 1297::I, 1301::I, _
1303::I, 1307::I, 1319::I, 1321::I, 1327::I, 1361::I, _
1367::I, 1373::I, 1381::I, 1399::I, 1409::I, 1423::I, _
1427::I, 1429::I, 1433::I, 1439::I, 1447::I, 1451::I, _
1453::I, 1459::I, 1471::I, 1481::I, 1483::I, 1487::I, _
1489::I, 1493::I, 1499::I, 1511::I, 1523::I, 1531::I, _
1543::I, 1549::I, 1553::I, 1559::I, 1567::I, 1571::I, _
1579::I, 1583::I, 1597::I, 1601::I, 1607::I, 1609::I, _
1613::I, 1619::I, 1621::I, 1627::I, 1637::I, 1657::I, _
1206

CHAPTER 10. CHAPTER I

1663::I, 1667::I, 1669::I, 1693::I, 1697::I, 1699::I,
1709::I, 1721::I, 1723::I, 1733::I, 1741::I, 1747::I,
1753::I, 1759::I, 1777::I, 1783::I, 1787::I, 1789::I,
1791::I, 1793::I, 1797::I, 1811::I, 1817::I, 1823::I,
1831::I, 1833::I, 1847::I, 1853::I, 1861::I, 1867::I,
1871::I, 1873::I, 1877::I, 1883::I, 1887::I, 1889::I,
1901::I, 1907::I, 1913::I, 1917::I, 1931::I, 1933::I,
1939::I, 1949::I, 1951::I, 1973::I, 1979::I, 1987::I,
1993::I, 1997::I, 1999::I, 2003::I, 2011::I, 2017::I,
2027::I, 2029::I, 2039::I, 2053::I, 2063::I, 2069::I,
2081::I, 2083::I, 2087::I, 2089::I, 2099::I, 2111::I,
2113::I, 2129::I, 2131::I, 2137::I, 2141::I, 2143::I,
2153::I, 2161::I, 2179::I, 2203::I, 2207::I, 2213::I,
2221::I, 2237::I, 2239::I, 2243::I, 2251::I, 2257::I,
2263::I, 2277::I, 2281::I, 2287::I, 2293::I, 2297::I,
2309::I, 2311::I, 2323::I, 2337::I, 2341::I, 2347::I,
2351::I, 2357::I, 2371::I, 2377::I, 2381::I, 2383::I,
2389::I, 2393::I, 2399::I, 2411::I, 2417::I, 2423::I,
2437::I, 2441::I, 2447::I, 2459::I, 2467::I, 2473::I,
2477::I, 2503::I, 2521::I, 2531::I, 2539::I, 2543::I,
2549::I, 2551::I, 2557::I, 2579::I, 2591::I, 2593::I,
2609::I, 2617::I, 2621::I, 2633::I, 2647::I, 2657::I,
2659::I, 2663::I, 2671::I, 2677::I, 2683::I, 2687::I,
2689::I, 2693::I, 2699::I, 2707::I, 2711::I, 2713::I,
2719::I, 2729::I, 2731::I, 2741::I, 2749::I, 2753::I,
2767::I, 2777::I, 2789::I, 2791::I, 2797::I, 2801::I,
2803::I, 2819::I, 2833::I, 2837::I, 2843::I, 2851::I,
2857::I, 2861::I, 2879::I, 2887::I, 2897::I, 2903::I,
2917::I, 2927::I, 2939::I, 2953::I, 2967::I, 2969::I,
2971::I, 2999::I, 3001::I, 3011::I, 3019::I, 3023::I,
3037::I, 3041::I, 3049::I, 3061::I, 3067::I, 3079::I,
3083::I, 3089::I, 3109::I, 3119::I, 3121::I, 3137::I,
3163::I, 3167::I, 3169::I, 3181::I, 3187::I, 3191::I,
3203::I, 3209::I, 3217::I, 3221::I, 3229::I, 3251::I,
3253::I, 3257::I, 3259::I, 3271::I, 3299::I, 3301::I,
3307::I, 3313::I, 3319::I, 3323::I, 3329::I, 3331::I,
3343::I, 3347::I, 3359::I, 3361::I, 3367::I, 3373::I,
3389::I, 3391::I, 3407::I, 3413::I, 3423::I, 3449::I,
3457::I, 3461::I, 3463::I, 3467::I, 3469::I, 3491::I,
3499::I, 3511::I, 3517::I, 3527::I, 3529::I, 3533::I,
3539::I, 3541::I, 3547::I, 3557::I, 3559::I, 3571::I,
3581::I, 3583::I, 3593::I, 3607::I, 3613::I, 3617::I,
3623::I, 3631::I, 3637::I, 3643::I, 3659::I, 3671::I,
3673::I, 3677::I, 3691::I, 3697::I, 3701::I, 3709::I,
3719::I, 3727::I, 3733::I, 3739::I, 3761::I, 3767::I,
3769::I, 3779::I, 3793::I, 3797::I, 3803::I, 3821::I,
3823::I, 3833::I, 3847::I, 3851::I, 3853::I, 3863::I,
3877::I, 3881::I, 3889::I, 3907::I, 3911::I, 3917::I,
3919::I, 3923::I, 3929::I, 3931::I, 3943::I, 3947::I,
3967::I, 3989::I, 4001::I, 4003::I, 4007::I, 4013::I,
4019::I, 4021::I, 4027::I, 4049::I, 4051::I, 4057::I,
4073::I, 4079::I, 4091::I, 4093::I, 4099::I, 4111::I,  
4127::I, 4129::I, 4133::I, 4139::I, 4153::I, 4157::I,  
4159::I, 4177::I, 4201::I, 4211::I, 4217::I, 4219::I,  
4229::I, 4241::I, 4243::I, 4253::I, 4259::I,  
4261::I, 4271::I, 4273::I, 4283::I, 4289::I, 4297::I,  
4327::I, 4337::I, 4339::I, 4349::I, 4357::I, 4363::I,  
4373::I, 4391::I, 4397::I, 4409::I, 4421::I, 4423::I,  
4441::I, 4447::I, 4451::I, 4457::I, 4463::I, 4481::I,  
4483::I, 4493::I, 4507::I, 4513::I, 4517::I, 4519::I,  
4523::I, 4547::I, 4549::I, 4561::I, 4567::I, 4583::I,  
4591::I, 4597::I, 4603::I, 4621::I, 4637::I, 4639::I,  
4643::I, 4649::I, 4651::I, 4657::I, 4663::I, 4673::I,  
4679::I, 4691::I, 4703::I, 4721::I, 4723::I, 4729::I,  
4733::I, 4751::I, 4759::I, 4783::I, 4787::I, 4789::I,  
4793::I, 4799::I, 4801::I, 4813::I, 4817::I, 4831::I,  
4861::I, 4871::I, 4877::I, 4889::I, 4903::I, 4909::I,  
4919::I, 4931::I, 4933::I, 4937::I, 4943::I, 4951::I,  
4957::I, 4967::I, 4969::I, 4973::I, 4987::I, 4993::I,  
4999::I, 5003::I, 5009::I, 5011::I, 5021::I, 5023::I,  
5039::I, 5051::I, 5059::I, 5077::I, 5081::I, 5087::I,  
5099::I, 5101::I, 5107::I, 5113::I, 5119::I, 5147::I,  
5153::I, 5167::I, 5171::I, 5179::I, 5189::I, 5197::I,  
5209::I, 5227::I, 5231::I, 5233::I, 5237::I, 5261::I,  
5273::I, 5279::I, 5281::I, 5287::I, 5303::I, 5309::I,  
5323::I, 5333::I, 5347::I, 5351::I, 5381::I, 5387::I,  
5393::I, 5399::I, 5407::I, 5413::I, 5417::I, 5419::I,  
5431::I, 5437::I, 5441::I, 5443::I, 5449::I, 5471::I,  
5477::I, 5479::I, 5483::I, 5501::I, 5503::I, 5507::I,  
5519::I, 5521::I, 5527::I, 5531::I, 5557::I, 5563::I,  
5569::I, 5573::I, 5581::I, 5591::I, 5623::I, 5639::I,  
5641::I, 5647::I, 5651::I, 5653::I, 5657::I, 5659::I,  
5669::I, 5683::I, 5689::I, 5693::I, 5701::I, 5711::I,  
5717::I, 5737::I, 5741::I, 5743::I, 5749::I, 5779::I,  
5783::I, 5791::I, 5801::I, 5807::I, 5813::I, 5821::I,  
5827::I, 5839::I, 5843::I, 5849::I, 5851::I, 5857::I,  
5861::I, 5867::I, 5869::I, 5879::I, 5881::I, 5897::I,  
5903::I, 5923::I, 5927::I, 5939::I, 5953::I, 5981::I,  
5987::I, 6007::I, 6011::I, 6029::I, 6037::I, 6043::I,  
6047::I, 6053::I, 6067::I, 6073::I, 6079::I, 6089::I,  
6091::I, 6101::I, 6113::I, 6121::I, 6131::I, 6133::I,  
6143::I, 6151::I, 6163::I, 6173::I, 6197::I, 6199::I,  
6203::I, 6211::I, 6217::I, 6221::I, 6229::I, 6247::I,  
6257::I, 6263::I, 6269::I, 6271::I, 6277::I, 6287::I,  
6299::I, 6301::I, 6311::I, 6317::I, 6323::I, 6329::I,  
6337::I, 6343::I, 6353::I, 6359::I, 6361::I, 6367::I,  
6373::I, 6379::I, 6389::I, 6397::I, 6421::I, 6427::I,  
6449::I, 6451::I, 6469::I, 6473::I, 6481::I, 6491::I,  
6521::I, 6529::I, 6547::I, 6551::I, 6553::I, 6563::I,  
6569::I, 6571::I, 6577::I, 6581::I, 6599::I, 6607::I,  
6619::I, 6637::I, 6653::I, 6659::I, 6661::I, 6673::I,
CHAPTER 10. CHAPTER I

6679::I, 6689::I, 6691::I, 6701::I, 6703::I, 6709::I,
6719::I, 6733::I, 6737::I, 6761::I, 6763::I, 6779::I,
6781::I, 6791::I, 6793::I, 6803::I, 6823::I, 6827::I,
6829::I, 6833::I, 6841::I, 6857::I, 6863::I, 6869::I,
6871::I, 6883::I, 6899::I, 6907::I, 6911::I, 6917::I,
6947::I, 6949::I, 6959::I, 6961::I, 6967::I, 6971::I,
6977::I, 6983::I, 6991::I, 6997::I, 7001::I, 7013::I,
7019::I, 7027::I, 7039::I, 7043::I, 7057::I, 7069::I,
7079::I, 7103::I, 7109::I, 7121::I, 7127::I, 7129::I,
7151::I, 7159::I, 7177::I, 7187::I, 7193::I, 7207::I,
7211::I, 7213::I, 7219::I, 7229::I, 7237::I, 7243::I,
7247::I, 7253::I, 7283::I, 7297::I, 7307::I, 7309::I,
7321::I, 7331::I, 7333::I, 7349::I, 7351::I, 7369::I,
7393::I, 7411::I, 7417::I, 7433::I, 7451::I, 7457::I,
7459::I, 7477::I, 7481::I, 7487::I, 7489::I, 7499::I,
7507::I, 7517::I, 7523::I, 7529::I, 7537::I, 7541::I,
7547::I, 7549::I, 7559::I, 7561::I, 7573::I, 7577::I,
7583::I, 7589::I, 7591::I, 7603::I, 7607::I, 7621::I,
7639::I, 7643::I, 7649::I, 7669::I, 7673::I, 7681::I,
7687::I, 7691::I, 7699::I, 7703::I, 7717::I, 7723::I,
7727::I, 7741::I, 7753::I, 7757::I, 7759::I, 7789::I,
7793::I, 7817::I, 7823::I, 7829::I, 7841::I, 7853::I,
7867::I, 7873::I, 7877::I, 7879::I, 7883::I, 7901::I,
7907::I, 7919::I, 7927::I, 7933::I, 7937::I, 7949::I,
7951::I, 7963::I, 7993::I, 8009::I, 8011::I, 8017::I,
8039::I, 8053::I, 8059::I, 8069::I, 8081::I, 8087::I,
8089::I, 8093::I, 8101::I, 8111::I, 8117::I, 8123::I,
8147::I, 8161::I, 8167::I, 8171::I, 8179::I, 8191::I,
8209::I, 8219::I, 8221::I, 8231::I, 8233::I, 8237::I,
8243::I, 8263::I, 8269::I, 8273::I, 8287::I, 8291::I,
8293::I, 8297::I, 8311::I, 8317::I, 8329::I, 8353::I,
8363::I, 8369::I, 8377::I, 8387::I, 8399::I, 8419::I,
8423::I, 8429::I, 8431::I, 8443::I, 8447::I, 8461::I,
8467::I, 8501::I, 8513::I, 8521::I, 8527::I, 8537::I,
8559::I, 8563::I, 8569::I, 8573::I, 8581::I, 8597::I,
8599::I, 8609::I, 8623::I, 8627::I, 8629::I, 8641::I,
8647::I, 8663::I, 8669::I, 8677::I, 8681::I, 8689::I,
8693::I, 8699::I, 8707::I, 8713::I, 8719::I, 8731::I,
8737::I, 8741::I, 8747::I, 8753::I, 8761::I, 8779::I,
8783::I, 8803::I, 8807::I, 8819::I, 8821::I, 8831::I,
8837::I, 8839::I, 8849::I, 8861::I, 8863::I, 8867::I,
8887::I, 8893::I, 8923::I, 8929::I, 8933::I, 8941::I,
8951::I, 8963::I, 8969::I, 8971::I, 8999::I, 9001::I,
9007::I, 9011::I, 9013::I, 9029::I, 9041::I, 9043::I,
9049::I, 9059::I, 9067::I, 9091::I, 9103::I, 9109::I,
9127::I, 9133::I, 9137::I, 9151::I, 9157::I, 9161::I,
9173::I, 9181::I, 9187::I, 9199::I, 9203::I, 9209::I,
9221::I, 9227::I, 9239::I, 9241::I, 9257::I, 9277::I,
9281::I, 9283::I, 9293::I, 9311::I, 9319::I, 9323::I,
9337::I, 9341::I, 9343::I, 9349::I, 9371::I, 9377::I,
primes

— package PRIMES IntegerPrimesPackage —

primes(m, n) ==
  -- computes primes from m to n inclusive using prime?
  1:List(I) :=
    m <= two => [two]
    empty()
n < two or n < m => empty()
if even? m then m := m + 1
ll:List(I) := [k::I for k in convert(m)@Integer..convert(n)@Integer by 2 | prime?(k::I)]
reverse_! concat_!(ll, l)

rabinProvesComposite : (I,I,I,I,NonNegativeInteger) -> Boolean
rabinProvesCompositeSmall : (I,I,I,I,NonNegativeInteger) -> Boolean

rabinProvesCompositeSmall
— package PRIMES IntegerPrimesPackage —

rabinProvesCompositeSmall(p,n,nm1,q,k) ==
  -- probability n prime is > 3/4 for each iteration
  -- for most n this probability is much greater than 3/4
  t := powmod(p, q, n)
  -- neither of these cases tells us anything
  if not ((t = 1) or t = nm1) then
    for j in 1..k-1 repeat
      oldt := t
      t := mulmod(t, t, n)
      (t = 1) => return true
      -- we have squared someting not -1 and got 1
      t = nm1 =>
        leave
      not (t = nm1) => return true
    false

rabinProvesComposite
— package PRIMES IntegerPrimesPackage —

rabinProvesComposite(p,n,nm1,q,k) ==
  -- probability n prime is > 3/4 for each iteration
  -- for most n this probability is much greater than 3/4
  t := powmod(p, q, n)
  -- neither of these cases tells us anything
if t=nm1 then count2Order(1):=count2Order(1)+1
if not ((t = 1) or t = nm1) then
  for j in 1..k-1 repeat
    oldt := t
    t := mulmod(t, t, n)
    (t = 1) => return true
    -- we have squared something not -1 and got 1
    t = nm1 =>
      rootsMinus1:=union(rootsMinus1,oldt)
      count2Order(j+1):=count2Order(j+1)+1
      leave
    not (t = nm1) => return true
  # rootsMinus1 > 2 => true -- Z/nZ can’t be a field
  false

prime?

— package PRIMES IntegerPrimesPackage —

prime? n ==
  n < two => false
  n < nextSmallPrime => member?(n, smallPrimes)
  not (gcd(n, productSmallPrimes) = 1) => false
  n < nextSmallPrimeSquared => true

  nm1 := n-1
  q := (nm1) quo two
  for k in 1.. while not odd? q repeat q := q quo two
  -- q = (n-1) quo 2**k for largest possible k

  n < JaeschkeLimit =>
    rabinProvesCompositeSmall(2::I,n,nm1,q,k) => return false
    rabinProvesCompositeSmall(3::I,n,nm1,q,k) => return false

  n < PomeranceLimit =>
    rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
    member?(n,PomeranceList) => return false
    true

    rabinProvesCompositeSmall(7::I,n,nm1,q,k) => return false

  n < PinchLimit =>
    rabinProvesCompositeSmall(10::I,n,nm1,q,k) => return false
    member?(n,PinchList) => return false
    true
rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
rabinProvesCompositeSmall(11::I,n,nm1,q,k) => return false
n < PinchLimit2 =>
    member?(n,PinchList2) => return false
true
rabinProvesCompositeSmall(13::I,n,nm1,q,k) => return false
rabinProvesCompositeSmall(17::I,n,nm1,q,k) => return false
true

rootsMinus1:= empty()
count2Order := new(k,0) -- vector of k zeroes

mn := minIndex smallPrimes
for i in mn+1..mn+10 repeat
    rabinProvesComposite(smallPrimes i,n,nm1,q,k) => return false
import IntegerRoots(I)
q > 1 and perfectSquare?(3*n+1) => false
((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare?(8*n+1) => false
-- Both previous tests from Damgard & Landrock
currPrime:=smallPrimes(mn+10)
probablySafe:=tenPowerTwenty
while count2Order(k) = 0 or n > probablySafe repeat
    currPrime := nextPrime currPrime
    probablySafe:=probablySafe*(100::I)
    rabinProvesComposite(currPrime,n,nm1,q,k) => return false
true

nextPrime

— package PRIMES IntegerPrimesPackage —

nextPrime n ==
    -- computes the first prime after n
    n < two => two
    if odd? n then n := n + two else n := n + 1
    while not prime? n repeat n := n + two
    n
prevPrime

— package PRIMES IntegerPrimesPackage —

prevPrime n ==
-- computes the first prime before n
n < 3::I => error "no primes less than 2"
n = 3::I => two
if odd? n then n := n - two else n := n - 1
while not prime? n repeat n := n - two
n

— PRIMES.dotabb —

"PRIMES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMES"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"PRIMES" -> "FSAGG"

package INTRET IntegerRetractions

— IntegerRetractions.input —

)set break resume
)sys rm -f IntegerRetractions.output
)spool IntegerRetractions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerRetractions
--E 1

)spool
)lisp (bye)

— IntegerRetractions.help —
IntegerRetractions examples

Provides integer testing and retraction functions.

See Also:
o )show IntegerRetractions

---

IntegerRetractions (INTRET)

Exports:
integer integer? integerIfCan

--- package INTRET IntegerRetractions ---

)abbrev package INTRET IntegerRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Provides integer testing and retraction functions.

IntegerRetractions(S:RetractableTo(Integer)): with
  integer : S -> Integer
    ++ integer(x) returns x as an integer;
    ++ error if x is not an integer;
  integer? : S -> Boolean
    ++ integer?(x) is true if x is an integer, false otherwise;
  integerIfCan: S -> Union(Integer, "failed")
    ++ integerIfCan(x) returns x as an integer,
++ "failed" if \( x \) is not an integer;
== add
integer \( s \) == retract \( s \)
integer? \( s \) == retractIfCan(\( s \)) case Integer
integerIfCan \( s \) == retractIfCan \( s \)

---

--- INTRET.dotabb ---

"INTRET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRET"]
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"INTRET" -> "RETRACT"

---

package IROOT IntegerRoots

--- IntegerRoots.input ---

)set break resume
)sys rm -f IntegerRoots.output
)spool IntegerRoots.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerRoots
--E 1

)spool
)lisp (bye)

---

--- IntegerRoots.help ---

====================================================================
IntegerRoots examples
====================================================================

The IntegerRoots package computes square roots and nth roots of
integers efficiently.

See Also:
- \texttt{show IntegerRoots}

---

**IntegerRoots (IROOT)**

Exports:

\begin{itemize}
  \item \texttt{approxSqrt}
  \item \texttt{approxNthRoot}
  \item \texttt{perfectNthPower?}
  \item \texttt{perfectNthRoot}
  \item \texttt{perfectSqrt}
  \item \texttt{perfectSquare?}
\end{itemize}

---

\texttt{\textbackslash abbrev package IROOT IntegerRoots}

++ Author: Michael Monagan
++ Date Created: November 1987
++ Description:
++ The \texttt{IntegerRoots} package computes square roots and
++ \texttt{n}th roots of integers efficiently.

\texttt{IntegerRoots(I:IntegerNumberSystem): Exports == Implementation where
  NNI \rightarrow \text{NonNegativeInteger}}

Exports == with

\begin{itemize}
  \item \texttt{perfectNthPower? (I, NNI) \rightarrow Boolean}
    \begin{itemize}
      \item \texttt{\textbackslash spad{perfectNthPower?(n,r)}} returns true if \texttt{n} is an \texttt{\textbackslash spad{r}}th power and false otherwise
    \end{itemize}
  \item \texttt{perfectNthRoot (I, NNI) \rightarrow \text{Union(I,\	extbackslash \textquoteleft\textbackslash failed\textquoteright)}}
    \begin{itemize}
      \item \texttt{\textbackslash spad{perfectNthRoot(n,r)}} returns the \texttt{\textbackslash spad{r}}th root of \texttt{n} if \texttt{n} is an \texttt{\textbackslash spad{r}}th power and returns "\texttt{failed}" otherwise
    \end{itemize}
  \item \texttt{perfectNthRoot: I \rightarrow Record(base:I, exponent:NNI)}
    \begin{itemize}
      \item \texttt{\textbackslash spad{perfectNthRoot(n)}} returns \texttt{\textbackslash spad{[x,r]}} where \texttt{\textbackslash spad{n = x^r}}
    \end{itemize}
\end{itemize}
++ and r is the largest integer such that n is a perfect \( \sqrt[r]{n} \)
approxNthRoot: (I,NNI) -> I
++ \( \text{approxNthRoot}(n,r) \) returns an approximation x
++ to \( n^{\frac{1}{r}} \) such that \( -1 < x - n^{\frac{1}{r}} < 1 \)
perfectSquare?: I -> Boolean
++ \( \text{perfectSquare?}(n) \) returns true if n is a perfect square
++ and false otherwise
perfectSqrt: I -> Union(I,"failed")
++ \( \text{perfectSqrt}(n) \) returns the square root of n if n is a
++ perfect square and returns "failed" otherwise
approxSqrt: I -> I
++ \( \text{approxSqrt}(n) \) returns an approximation x
++ to \( \sqrt{n} \) such that \( -1 < x - \sqrt{n} < 1 \).
++ Compute an approximation s to \( \sqrt{n} \) such that
++ \( -1 < s - \sqrt{n} < 1 \)
++ A variable precision Newton iteration is used.
++ The running time is \( O(\log(n)^2) \).

Implementation ==> add
import IntegerPrimesPackage(I)

resMod144: List I := [0::I,1::I,4::I,9::I,16::I,25::I,36::I,49::I,_
52::I,64::I,73::I,81::I,97::I,100::I,112::I,121::I]
two := 2::I

---

perfectSquare?

---

perfectNthPower?

---
perfectNthRoot

--- package IROOT IntegerRoots ---

perfectNthRoot n == -- complexity (log log n)**2 (log n)**2
m:NNI
(n = 1) or zero? n or n = -1 => [n, 1]
e:NNI := 1
p:NNI := 2
while p::I <= length(n) + 1 repeat
  for m in 0.. while (r := perfectNthRoot(n, p)) case I repeat
    n := r::I
    e := e * p ** m
    p := convert(nextPrime(p::I))@Integer :: NNI
  [n, e]

approxNthRoot

--- package IROOT IntegerRoots ---

approxNthRoot(a, n) == -- complexity (log log n) (log n)**2
zero? n => error "invalid arguments"
(n = 1) => a
n=2 => approxSqrt a
negative? a =>
  odd? n => - approxNthRoot(-a, n)
  0
zero? a => 0
(a = 1) => 1
-- quick check for case of large n
((3*n) quo 2)::I >= (l := length a) => two
-- the initial approximation must be >= the root
y := max(two, shift(1, (n::I+l-1) quo (n::I)))
z:1 := 1
n1:= (n-1)::NNI
while z > 0 repeat
  x := y
  xn:= x**n1
  y := (n1*x*xn+a) quo (n*xn)
  z := x-y
x
perfectNthRoot

--- package IROOT IntegerRoots ---

perfectNthRoot(b, n) ==
(r := approxNthRoot(b, n)) ** n = b => r
"failed"

perfectSqrt

--- package IROOT IntegerRoots ---

perfectSqrt a ==
a < 0 or not member?(a rem (144::I), resMod144) => "failed"
(s := approxSqrt a) * s = a => s
"failed"

approxSqrt

--- package IROOT IntegerRoots ---

approxSqrt a ==
a < 1 => 0
if (n := length a) > (100::I) then
  -- variable precision newton iteration
  n := n quo (4::I)
  s := approxSqrt shift(a, -2 * n)
  s := shift(s, n)
  return ((1 + s + a quo s) quo two)
-- initial approximation for the root is within a factor of 2
(new, old) := (shift(1, n quo two), 1)
while new ^= old repeat
  (new, old) := ((1 + new + a quo new) quo two, new)
new
package INTSLPE IntegerSolveLinearPolynomialEquation

--- IntegerSolveLinearPolynomialEquation.input ---

)set break resume
)sys rm -f IntegerSolveLinearPolynomialEquation.output
)spool IntegerSolveLinearPolynomialEquation.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show IntegerSolveLinearPolynomialEquation
-- E 1

)spool
)lisp (bye)

---

--- IntegerSolveLinearPolynomialEquation.help ---

====================================================================
IntegerSolveLinearPolynomialEquation examples
====================================================================

This package provides the implementation for the
solveLinearPolynomialEquation operation over the integers.
It uses a lifting technique from the package GenExEuclid.

See Also:
o )show IntegerSolveLinearPolynomialEquation
IntegerSolveLinearPolynomialEquation (INTSLPE)

Exports:
solveLinearPolynomialEquation

— package INTSLPE IntegerSolveLinearPolynomialEquation —

)abbrev package INTSLPE IntegerSolveLinearPolynomialEquation
++ Author: Davenport
++ Date Created: 1991
++ Description:
++ This package provides the implementation for the
++ \spadfun{solveLinearPolynomialEquation}
++ operation over the integers. It uses a lifting technique
++ from the package GenExEuclid

IntegerSolveLinearPolynomialEquation(): C ==
where
  ZP ==> SparseUnivariatePolynomial Integer
  C == with
    solveLinearPolynomialEquation: (List ZP,ZP) -> Union(List ZP,"failed")
    ++ solveLinearPolynomialEquation([f1, ..., fn], g)
    ++ (where the fi are relatively prime to each other)
    ++ returns a list of ai such that
    ++ \spad{g/prod fi = sum ai/fi}
    ++ or returns "failed" if no such list of ai’s exists.
  T == add
    oldlp:List ZP := []
    slpePrime:Integer:=(2::Integer)
    oldtable:Vector List ZP := empty()
    solveLinearPolynomialEquation(lp,p) ==
      if (oldlp ^= lp) then
        -- we have to generate a new table
        deg:= +/[degree u for u in lp]
        ans:=Union(Vector List ZP,"failed")::"failed"
        slpePrime:=2147483647::Integer -- 2**31 -1 : a prime
        -- a good test case for this package is
-- ([x**31-1,x-2],2)
while (ans case "failed") repeat
  ans:=tablePow(deg,slpePrime,lp)$GenExEuclid(Integer,ZP)
  if (ans case "failed") then
    slpePrime:= prevPrime(slpePrime)$IntegerPrimesPackage(Integer)
  
  oldtable:=(ans:: Vector List ZP)
  answer:=solveid(p,slpePrime,oldtable)

answer

——

— INTSLPE.dotabb —

"INTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTSLPE" -> "PFECAT"

——

package IBATOOL integralBasisTools

— IntegralBasisTools.input —

)set break resume
)sys rm -f IntegralBasisTools.output
)spool IntegralBasisTools.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show IntegralBasisTools
-- E 1

)spool
)lisp (bye)

——

— IntegralBasisTools.help —

====================================================================
IntegralBasisTools examples
This package contains functions used in the packages
FunctionFieldIntegralBasis and NumberFieldIntegralBasis.

See Also:
o )show IntegralBasisTools

---

IntegralBasisTools (IBATOOL)

Exports:
diagonalProduct  divideIfCan!  idealiser  idealiserMatrix  leastPower
matrixGcd        moduleSum

--- package IBATOOL IntegralBasisTools ---

)abbrev package IBATOOL IntegralBasisTools
++ Author: Victor Miller, Barry Trager, Clifton Williamson
++ Date Created: 11 April 1990
++ Date Last Updated: 20 September 1994
++ Description:
++ This package contains functions used in the packages
++ FunctionFieldIntegralBasis and NumberFieldIntegralBasis.

IntegralBasisTools(R,UP,F): Exports == Implementation where
  R : EuclideanDomain with
    squareFree: $ -> Factored $
    ++ squareFree(x) returns a square-free factorisation of x
  UP : UnivariatePolynomialCategory R
  F : FramedAlgebra(R,UP)
  Mat ==> Matrix R
  NNI ==> NonNegativeInteger
Ans ==> Record(basis: Mat, basisDen: R, basisInv: Mat)

Exports ==> with

  diagonalProduct: Mat -> R
  ++ diagonalProduct(m) returns the product of the elements on the
  ++ diagonal of the matrix m

matrixGcd: (Mat,R,NNI) -> R
  ++ matrixGcd(mat,sing,n) is \texttt{gcd(sing,g)} where \texttt{gcd(g)} is the
  ++ gcd of the entries of the \texttt{spad(n)}-by-\texttt{spad(n)} upper-triangular
  ++ matrix \spad{mat}.

divideIfCan_!: (Matrix R,Matrix R,R,Integer) -> R
  ++ divideIfCan!(matrix,matrixOut,prime,n) attempts to divide the
  ++ entries of \spad{matrix} by \spad{prime} and store the result in
  ++ \spad{matrixOut}. If it is successful, 1 is returned and if not,
  ++ \spad{prime} is returned. Here both \spad{matrix} and
  ++ \spad{matrixOut} are \spad{n}-by-\spad{n} upper triangular matrices.

leastPower: (NNI,NNI) -> NNI
  ++ leastPower(p,n) returns e, where e is the smallest integer
  ++ such that \spad{p ** e >= n}

idealiser: (Mat,Mat) -> Mat
  ++ idealiser(m1,m2) computes the order of an ideal defined by m1 and m2

idealiser: (Mat,Mat,R) -> Mat
  ++ idealiser(m1,m2,d) computes the order of an ideal defined by m1 and m2
  ++ where d is the known part of the denominator

idealiserMatrix: (Mat, Mat) -> Mat
  ++ idealiserMatrix(m1, m2) returns the matrix representing the linear
  ++ conditions on the Ring associated with an ideal defined by m1 and m2.

moduleSum: (Ans,Ans) -> Ans
  ++ moduleSum(m1,m2) returns the sum of two modules in the framed
  ++ algebra \spad{F}. Each module \spad{mi} is represented as follows:
  ++ \spad{F} is a framed algebra with \spad{R}-module basis \spad{w1,w2,...,wn} and
  ++ \spad{spad(mi)} is a record \spad{[basis,basisDen,basisInv]}. If
  ++ \spad{spad(basis)} is the matrix \spad{[aij, i = 1..n, j = 1..n]}, then
  ++ a basis \spad{v1,...,vn} for \spad{spad(mi)} is given by
  ++ \spad{spad(vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
  ++ \spad{spad(i)}th row of \spad{basis} contains the coordinates of the
  ++ \spad{spad(i)}th basis vector. Similarly, the \spad{spad(i)}th row of the
  ++ matrix \spad{spad(basisInv)} contains the coordinates of \spad{spad(wi)} with
  ++ respect to the basis \spad{spad(v1,...,vn}): if \spad{spad(basisInv)} is the
  ++ matrix \spad{spad((bij, i = 1..n, j = 1..n))}, then
  ++ \spad{spad(wi = sum(bij * vj, j = 1..n))}.

Implementation ==> add
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

diagonalProduct m ==
  ans : R := 1
  for i in minRowIndex m .. maxRowIndex m
for j in minColIndex m .. maxColIndex m repeat
  ans := ans * qelt(m, i, j)

matrixGcd(mat, sing, n) ==
  -- note that 'matrix' is upper triangular;
  -- no need to do anything below the diagonal
  d := sing
  for i in 1..n repeat
    for j in i..n repeat
      if not zero?(mij := qelt(mat, i, j)) then d := gcd(d, mij)
  -- one? d => return d
  (d = 1) => return d
  d

divideIfCan!(matrix, matrixOut, prime, n) ==
  -- note that both 'matrix' and 'matrixOut' will be upper triangular;
  -- no need to do anything below the diagonal
  for i in 1..n repeat
    for j in i..n repeat
      (a := (qelt(matrix, i, j) exquo prime)) case "failed" => return prime
      qsetelt_!(matrixOut, i, j, a :: R)
  1

leastPower(p, n) ==
  -- efficiency is not an issue here
  e : NNI := 1; q := p
  while q < n repeat (e := e + 1; q := q * p)
  e

idealiserMatrix(ideal, idealinv) ==
  -- computes the Order of the ideal
  n := rank()$F
  bigm := zero(n * n, n)$Mat
  mr := minRowIndex bigm; mc := minColIndex bigm
  v := basis()$F
  for i in 0..n-1 repeat
    r := regularRepresentation qelt(v, i + minIndex v)
    m := ideal * r * idealinv
    for j in 0..n-1 repeat
      for k in 0..n-1 repeat
        bigm(j * n + k + mr, i + mc) := qelt(m, j + mr, k + mc)

idealiser(ideal, idealinv) ==
  bigm := idealiserMatrix(ideal, idealinv)
  transpose squareTop rowEch bigm

idealiser(ideal, idealinv, denom) ==
  bigm := (idealiserMatrix(ideal, idealinv) exquo denom)::Mat
transpose squareTop rowEchelon(bigm,denom)

moduleSum(mod1,mod2) ==
  rb1 := mod1.basis; rbden1 := mod1.basisDen; rbinv1 := mod1.basisInv
  rb2 := mod2.basis; rbden2 := mod2.basisDen; rbinv2 := mod2.basisInv
  -- compatibility check: doesn't take much computation time
  (not square? rb1) or (not square? rbinv1) or (not square? rb2) _
  or (not square? rbinv2) =>
    error "moduleSum: matrices must be square"
  (n := nrows rb1) ^= (nrows rbinv1)) or (n ^= (nrows rb2)) _
  or (n ^= (nrows rbinv2)) =>
    error "moduleSum: matrices of incompatible dimensions"
  (zero? rbden1) or (zero? rbden2) =>
    error "moduleSum: denominator must be non-zero"
  den := lcm(rbden1,rbden2); c1 := den quo rbden1; c2 := den quo rbden2
  rb := squareTop rowEchelon(vertConcat(c1 * rb1,c2 * rb2),den)
  rbinv := UpTriBddDenomInv(rb,den)
  [rb,den,rbinv]

— IBATOOL.dotabb —

"IBATOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBATOOL"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"IBATOOL" -> "FRAMALG"

— package IBPTOOLS IntegralBasisPolynomialTools —

package IBPTOOLS IntegralBasisPolynomialTools

  — IntegralBasisPolynomialTools.input —

)set break resume
)sys rm -f IntegralBasisPolynomialTools.output
)spool IntegralBasisPolynomialTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegralBasisPolynomialTools
--E 1
IntegralBasisPolynomialTools provides functions for mapping functions on the coefficients of univariate and bivariate polynomials.

See Also:
o )show IntegralBasisPolynomialTools

Exports:
mapBivariate mapMatrixIfCan mapUnivariate mapUnivariateIfCan

package IBPTOOLS IntegralBasisPolynomialTools

 IntegralBasisPolynomialTools (IBPTOOLS)
on the coefficients of univariate and bivariate polynomials.

IntegralBasisPolynomialTools(K,R,UP,L): Exports == Implementation where
  K : Ring
  R : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R
  L : Ring

  MAT ==> Matrix
  SUP ==> SparseUnivariatePolynomial

Exports ==> with
  mapUnivariate: (L -> K,SUP L) -> R
  ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}.

  mapUnivariate: (K -> L,R) -> SUP L
  ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}.

  mapUnivariateIfCan: (L -> Union(K,"failed"),SUP L) -> Union(R,"failed")
  ++ mapUnivariateIfCan(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}, if possible, and returns
  ++ \spad{"failed"} otherwise.

  mapMatrixIfCan: (L -> Union(K,"failed"),MAT SUP L) -> Union(MAT R,"failed")
  ++ mapMatrixIfCan(f,mat) applies the function \spad{f} to the
  ++ coefficients of the entries of \spad{mat} if possible, and returns
  ++ \spad{"failed"} otherwise.

  mapBivariate: (K -> L,UP) -> SUP SUP L
  ++ mapBivariate(f,p(x,y)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x,y)}.

Implementation ==> add

mapUnivariate(f:L -> K,poly:SUP L) ==
  ans : R := 0
  while not zero? poly repeat
    ans := ans + monomial(f leadingCoefficient poly,degree poly)
    poly := reductum poly
  ans

mapUnivariate(f:K -> L,poly:R) ==
  ans : SUP L := 0
  while not zero? poly repeat
    ans := ans + monomial(f leadingCoefficient poly,degree poly)
    poly := reductum poly
  ans
mapUnivariateIfCan(f,poly) ==
ans : R := 0
while not zero? poly repeat
  (lc := f leadingCoefficient poly) case "failed" => return "failed"
  ans := ans + monomial(lc :: K,degree poly)
poly := reductum poly
ans

mapMatrixIfCan(f,mat) ==
m := nrows mat; n := ncols mat
matOut : MAT R := new(m,n,0)
for i in 1..m repeat for j in 1..n repeat
  (poly := mapUnivariateIfCan(f,qelt(mat,i,j))) case "failed" =>
    return "failed"
  qsetelt_!(matOut,i,j,poly :: R)
matOut

mapBivariate(f,poly) ==
ans : SUP SUP L := 0
while not zero? poly repeat
  ans :=
    ans + monomial(mapUnivariate(f,leadingCoefficient poly),degree poly)
poly := reductum poly
ans

— IBPTOOLS.dotabb —

"IBPTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBPTOOLS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"IBPTOOLS" -> "PFECAT"

package IR2 IntegrationResultFunctions2

— IntegrationResultFunctions2.input —

)set break resume
)sys rm -f IntegrationResultFunctions2.output
)spool IntegrationResultFunctions2.output
)set message test on
)set message auto off
IntegrationResultFunctions2 (IR2)

Exports:
map

— package IR2 IntegrationResultFunctions2 —

)abbrev package IR2 IntegrationResultFunctions2
++ Author: Manuel Bronstein  
++ Date Created: 1987  
++ Date Last Updated: 12 August 1992  
++ Description:  
++ Internally used by the integration packages

IntegrationResultFunctions2(E, F): Exports == Implementation where  
E : Field  
F : Field

SE ==> Symbol  
Q ==> Fraction Integer  
IRE ==> IntegrationResult E  
IRF ==> IntegrationResult F  
UPE ==> SparseUnivariatePolynomial E  
UPF ==> SparseUnivariatePolynomial F  
NEE ==> Record(integrand:E, intvar:E)  
NEF ==> Record(integrand:F, intvar:F)  
LGE ==> Record(scalar:Q, coeff:UPE, logand:UPE)  
LGF ==> Record(scalar:Q, coeff:UPF, logand:UPF)  
NLE ==> Record(coeff:E, logand:E)  
NLF ==> Record(coeff:F, logand:F)  
UFE ==> Union(Record(mainpart:E, limitedlogs:List NLE), "failed")  
URE ==> Union(Record(ratpart:E, coeff:E), "failed")  
UE ==> Union(E, "failed")

Exports ==> with  
map: (E -> F, IRE) -> IRF  
++ map(f,ire) \ undocumented  
map: (E -> F, URE) -> Union(Record(ratpart:F, coeff:F), "failed")  
++ map(f,ure) \ undocumented  
map: (E -> F, UE) -> Union(F, "failed")  
++ map(f,ue) \ undocumented  
map: (E -> F, UFE) ->  
Union(Record(mainpart:F, limitedlogs:List NLF), "failed")  
++ map(f,ufe) \ undocumented

Implementation ==> add  
import SparseUnivariatePolynomialFunctions2(E, F)

NEE2F: (E -> F, NEE) -> NEF  
LGE2F: (E -> F, LGE) -> LGF  
NLE2F: (E -> F, NLE) -> NLF

NLE2F(func, r) == [func(r.coeff), func(r.logand)]  
NEE2F(func, n) == [func(n.integrand), func(n.intvar)]  
map(func:E -> F, u:UE) == (u case "failed" => "failed"; func(u::E))

map(func:E -> F, ir:IRE) ==  
mkAnswer(func ratpart ir, [LGE2F(func, f) for f in logpart ir],
map(func:E -> F, u:URE) ==
    u case "failed" => "failed"
    [func(u.ratpart), func(u.coeff)]

map(func:E -> F, u:UFE) ==
    u case "failed" => "failed"
    [func(u.mainpart), [NLE2F(func, f) for f in u.limitedlogs]]

LGE2F(func, lg) ==
    [lg.scalar, map(func, lg.coeff), map(func, lg.logand)]
IntegrationResultRFToFunction examples

Conversion of integration results to top-level expressions. This package allows a sum of logs over the roots of a polynomial to be expressed as explicit logarithms and arc tangents, provided that the indexing polynomial can be factored into quadratics.

See Also:
o )show IntegrationResultRFToFunction

IntegrationResultRFToFunction (IRRF2F)

Exports:
complexExpand complexIntegrate expand integrate split

package IRRF2F IntegrationResultRFToFunction

IntegrationResultRFToFunction(R):Exports == Implementation where
R:Join(GcdDomain,RetractableTo Integer,OrderedSet,
LinearlyExplicitRingOver Integer)

RF ==> Fraction Polynomial R
F ==> Expression R
IR ==> IntegrationResult RF

Exports ==> with
  split : IR -> IR
    ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
    ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + ... + sum_{Pn(a)=0} Q(a,x)}
    ++ where P1,...,Pn are the factors of P.
  expand : IR -> List F
    ++ expand(i) returns the list of possible real functions
    ++ corresponding to i.
  complexExpand : IR -> F
    ++ complexExpand(i) returns the expanded complex function
    ++ corresponding to i.
  if R has CharacteristicZero then
    integrate : (RF, Symbol) -> Union(F, List F)
      ++ integrate(f, x) returns the integral of \spad{f(x)dx}
      ++ where x is viewed as a real variable..
    complexIntegrate : (RF, Symbol) -> F
      ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
      ++ where x is viewed as a complex variable.

Implementation ==> add
  import IntegrationTools(R, F)
  import TrigonometricManipulations(R, F)
  import IntegrationResultToFunction(R, F)

  toEF: IR -> IntegrationResult F
  toEF i == map(z1+->z1::F, i)$IntegrationResultFunctions2(RF, F)
  expand i == expand toEF i
  complexExpand i == complexExpand toEF i
  split i ==
    map(retract, split toEF i)$IntegrationResultFunctions2(F, RF)

  if R has CharacteristicZero then
    import RationalFunctionIntegration(R)
    complexIntegrate(f, x) == complexExpand internalIntegrate(f, x)

    -- do not use real integration if R is complex
    if R has imaginary: () -> R then integrate(f, x) == complexIntegrate(f, x)
    else
      integrate(f, x) ==
      l := [mkPrim(real g, x) for g in expand internalIntegrate(f, x)]
      empty? rest l => first l
package IR2F IntegrationResultToFunction

IntegrationResultToFunction examples
====================================================================
Conversion of integration results to top-level expressions
This package allows a sum of logs over the roots of a polynomial to be expressed as explicit logarithms and arc tangents, provided that the indexing polynomial can be factored into quadratics.
See Also:
- )show IntegrationResultToFunction

---

IntegrationResultToFunction (IR2F)

IR2F

↓

ACFS

Exports:
- complexExpand
- expand
- split

--- package IR2F IntegrationResultToFunction ---

)abbrev package IR2F IntegrationResultToFunction
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 9 October 1991
++ Description:
++ Conversion of integration results to top-level expressions
++ This package allows a sum of logs over the roots of a polynomial
++ to be expressed as explicit logarithms and arc tangents, provided
++ that the indexing polynomial can be factored into quadratics.

IntegrationResultToFunction(R, F): Exports == Implementation where
  R: Join(GcdDomain, RetractableTo Integer, OrderedSet,
         LinearlyExplicitRingOver Integer)
  F: Join(AlgebraicallyClosedFunctionSpace R,
         TranscendentalFunctionCategory)
  N  ==> NonNegativeInteger
  Z  ==> Integer
  Q  ==> Fraction Z
  K  ==> Kernel F
  P  ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
IR ==> IntegrationResult F
REC ==> Record(ans1:F, ans2:F)
LOG ==> Record(scalar:Q, coeff:UP, logand:UP)

Exports ==> with
  split : IR -> IR
    ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
    ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + ... + sum_{Pn(a)=0} Q(a,x)}
    ++ where P1,...,Pn are the factors of P.
  expand : IR -> List F
    ++ expand(i) returns the list of possible real functions
    ++ corresponding to i.
  complexExpand : IR -> F
    ++ complexExpand(i) returns the expanded complex function
    ++ corresponding to i.

Implementation ==> add
  import AlgebraicManipulations(R, F)
  import ElementaryFunctionSign(R, F)

IR2F : IR -> F
ir2f i == retract mkAnswer(ratpart i, empty(), notelem i)

pairprod(x, l) == [x * y for y in l]
evenRoots x ==
  [first argument k for k in tower x |
   is?(k,"nthRoot"::Symbol) and even?(retract(second argument k)@Z)
   and (not empty? variables first argument k)]
j := split i
pairsum([[IR2F j], mkRealFunc logpart j])

split i ==
  mkAnswer(ratpart i, concat [nlogs l for l in logpart i], notelem i)

complexExpand i ==
  j := split i
  IR2F j + +/[lg.scalar::F * lg2cfunc lg for lg in logpart j]

-- p = a t^2 + b t + c
-- Expands sum_{p(t) = 0} t log(lg(t))
quadratic(p, lg) ==
  zero?(delta := (b := coefficient(p, 1))**2 - 4 * (a := coefficient(p, 2)) * (p0 := coefficient(p, 0))) =>
    [linear(monomial(1, 1) + (b / a)::UP, lg)]
  e := (q := quadeval(lg, c := - b * (d := inv(2*a)), d, delta)).ans1
  lgp := c * log(nrm := (e**2 - delta * (f := q.ans2)**2))
  s := (sqr := insqrt delta).sqrt
  pp := nn := 0$F
  if sqr.sgn >= 0 then
    sqrp := s.ans1 * rootSimp sqrt(s.ans2)
    pp := lgp + d * sqrp * log(((2 * e * f) / nrm) * sqrp
      + (e**2 + delta * f**2) / nrm)
  if sqr.sgn <= 0 then
    sqrn := s.ans1 * rootSimp sqrt(-s.ans2)
    nn := lgp + d * sqrn * ilog(e, f * sqrn,
      setUnion(setUnion(kernels a, kernels b), kernels p0))
  if sqr.sgn > 0 => [pp]
  if sqr.sgn < 0 => [nn]
  [pp, nn]

-- returns 2 atan(a/b) or 2 atan(-b/a) whichever looks better
-- they differ by a constant so it's ok to do it from an IR
tantrick(a, b) ==
  retractIfCan(a)@Union(Q, "failed") case Q => 2 * atan(-b/a)
  2 * atan(a/b)

-- transforms i log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo's algorithm
ilog(a, b, lk) ==
  l := setDifference(setUnion(variables numer a, variables numer b),
    setUnion(lk, setUnion(variables denom a, variables denom b)))
  empty? l => tantrick(a, b)
  k := "max"/l
  ilog0(a, b, numer univariate(a, k), numer univariate(b, k), k::F)

-- transforms i log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo's algorithm
-- the arc-tangents will not have k in the denominator
-- we always keep upa(k) = a and upb(k) = b
ilog0(a, b, upa, upb, k) ==
  if degree(upa) < degree(upb) then
    (upa, upb) := (-upb, upa)
    (a, b) := (-b, a)
  zero? degree upb => tantrick(a, b)
  r := extendedEuclidean(upa, upb)
  (g:= retractIfCan(r.generator)@Union(F,"failed")) case "failed" =>
    tantrick(a, b)
  if degree(r.coef1) >= degree upb then
    qr := divide(r.coef1, upb)
    r.coef1 := qr.remainder
    r.coef2 := r.coef2 + qr.quotient * upa
  aa := (r.coef2) k
  bb := -(r.coef1) k
  tantrick(aa * a + bb * b, g::F) + ilog0(aa,bb,r.coef2,-r.coef1,k)

lg2func lg ==
  zero?(d := degree(p := lg.coeff)) => error "poly has degree 0"
  one? d => [linear(p, lg.logand)]
  (d = 1) => [linear(p, lg.logand)]
  d = 2 => quadratic(p, lg.logand)
  odd? d and
    ((r := retractIfCan(reductum p)@Union(F, "failed")) case F) =>
      pairsum([cmplex(alpha := rootSimp zeroOf p, lg.logand)],
               lg2func [lg.scalar,
                        (p exquo (monomial(1, 1)$UP - alpha::UP))::UP,
                        lg.logand])
  [lg2cfunc lg]

lg2cfunc lg ==
  +/[cmplex(alpha, lg.logand) for alpha in zerosOf(lg.coeff)]

mkRealFunc l ==
  ans := empty()$List(F)
  for lg in l repeat
    ans := pairsum(ans, pairprod(lg.scalar::F, lg2func lg))
  ans

-- returns a log(b)
linear(p, lg) ==
  alpha := - coefficient(p, 0) / coefficient(p, 1)
  alpha * log lg alpha

-- returns (c, d) s.t. p(a + b t) = c + d t, where t^2 = delta
quadeval(p, a, b, delta) ==
  zero? p => [0, 0]
  bi := c := d := 0$F
  ai := 1$F
\[ v := \text{vectorise}(p, 1 + \text{degree} \ p) \]

\[
\begin{align*}
&\text{for } i \text{ in } \text{minIndex } v .. \text{maxIndex } v \text{ repeat} \\
&\quad c := c + \text{qelt}(v, i) \times ai \\
&\quad d := d + \text{qelt}(v, i) \times bi \\
&\quad \text{temp} := a \times ai + b \times bi \times \text{delta} \\
&\quad bi := a \times bi + b \times ai \\
&\quad ai := \text{temp} \\
&\quad [c, d]
\end{align*}
\]

\[
\text{compatible?}(lx, ly) = \\
\text{empty? } ly \Rightarrow true \\
\text{for } x \text{ in } lx \text{ repeat} \\
\quad \text{for } y \text{ in } ly \text{ repeat} \\
\quad \quad ((s := \text{sign}(x \times y)) \text{ case } Z) \text{ and } (s::Z < 0) \Rightarrow \text{return false} \\
\quad \text{true}
\]

\[
\text{pairsum}(lx, ly) = \\
\text{empty? } lx \Rightarrow ly \\
\text{empty? } ly \Rightarrow lx \\
\text{l} := \text{empty()}$\text{List}(F) \\
\text{for } x \text{ in } lx \text{ repeat} \\
\quad ls := \text{evenRoots} x \\
\quad \text{if not empty?}(ln := \\
\text{[x + y for } y \text{ in } ly \mid \text{compatible?}(ls, \text{evenRoots} y)) \text{ then} \\
\quad \quad l := \text{removeDuplicates} \text{concat}(l, ln) \\
\text{l}
\]

\[ -- \text{returns } [[a, b], s] \text{ where } \sqrt(y) = a \sqrt(b) \text{ and} \]
\[ -- s = 1 \text{ if } b > 0, -1 \text{ if } b < 0, 0 \text{ if the sign of } b \text{ cannot be determined} \]

\[
\text{insqrt} y = \\
\text{rec} := \text{froot}(y, 2)\text{\$PolynomialRoots(IndexedExponents } K, K, R, P, F) \\
\text{-- one?}(\text{rec.exponent}) \Rightarrow [[\text{rec.coef } \times \text{rec.radicand}, 1], 1] \\
((\text{rec.exponent}) = 1) \Rightarrow [[\text{rec.coef } \times \text{rec.radicand}, 1], 1] \\
\text{rec.exponent } ^= 2 \Rightarrow \text{error } "\text{Should not happen}" \\
\quad [[\text{rec.coef}, \text{rec.radicand}], \\
\quad \text{((s := \text{sign}(\text{rec.radicand})) \text{ case } "\text{failed}" \Rightarrow 0; s::Z)]]
\]

\[
\text{nlogs} \ lg = \\
[\text{[f.exponent } \times \text{lg.scalar}, \text{f.factor}, \text{lg.logand}] \text{ for } f \text{ in factors } \\
\text{ffactor(primitivePart(lg.coeff) \\
\quad )\$FunctionSpaceUnivariatePolynomialFactor(R, F, UP)]}
\]

\[\text{--------}\]

\[\text{-- IR2F.dotabb}\]

"IR2F" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IR2F"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"IR2F" -> "ACFS"

package INTTOOLS IntegrationTools

---

IntegrationTools.input

```lisp
(set break resume
(sys rm -f IntegrationTools.output
(spool IntegrationTools.output
(set message test on
(set message auto off
(clear all

--S 1 of 1
(show IntegrationTools

--E 1

(spool
(lisp (bye)
```

---

IntegrationTools.help

====================================================================
IntegrationTools examples
====================================================================

Tools for the integrator

See Also:
 o (show IntegrationTools

---
IntegrationTools (INTTOOLS)

Exports:

\begin{itemize}
  \item \texttt{kmax}
  \item \texttt{intPatternMatch}
  \item \texttt{ksec}
  \item \texttt{mkPrim}
  \item \texttt{removeConstantTerm}
  \item \texttt{union}
  \item \texttt{vark}
  \item \texttt{varselect}
\end{itemize}

\begin{verbatim}
— package INTTOOLS IntegrationTools —

)abbrev package INTTOOLS IntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 25 April 1990
++ Date Last Updated: 9 June 1993
++ Description:
++ Tools for the integrator

IntegrationTools(R:OrderedSet, F:FunctionSpace R): Exp == Impl where
  K ==> Kernel F
  SE ==> Symbol
  P ==> SparseMultivariatePolynomial(R, K)
  UP ==> SparseUnivariatePolynomial F
  IR ==> IntegrationResult F
  ANS ==> Record(special:F, integrand:F)
  U ==> Union(ANS, "failed")
  ALGOP ==> "%alg"

Exp == with
  varselect: (List K, SE) -> List K
    ++ varselect([k1,...,kn], x) returns the ki which involve x.
  kmax : List K -> K
    ++ kmax([k1,...,kn]) returns the top-level ki for integration.
  ksec : (K, List K, SE) -> K
    ++ ksec(k, [k1,...,kn], x) returns the second top-level ki
    ++ after k involving x.
  union : (List K, List K) -> List K
    ++ union(l1, l2) returns set-theoretic union of l1 and l2.
  vark : (List F, SE) -> List K
    ++ vark([f1,...,fn],x) returns the set-theoretic union of
\end{verbatim}
++ \spad{(varselect(f1,x),...,varselect(fn,x))}.
if R has IntegralDomain then
  removeConstantTerm: (F, SE) -> F
  ++ removeConstantTerm(f, x) returns f minus any additive constant
  ++ with respect to x.
if R has GcdDomain and F has ElementaryFunctionCategory then
  mkPrim: (F, SE) -> F
  ++ mkPrim(f, x) makes the logs in f which are linear in x
  ++ primitive with respect to x.
if R has ConvertibleTo Pattern Integer and R has PatternMatchable Integer
  and F has LiouvillianFunctionCategory and F has RetractableTo SE then
  intPatternMatch: (F, SE, (F, SE) -> IR, (F, SE) -> U) -> IR
  ++ intPatternMatch(f, x, int, pmint) tries to integrate \spad{f}
  ++ first by using the integration function \spad{int}, and then
  ++ by using the pattern match intetgration function \spad{pmint}
  ++ on any remaining unintegrable part.

Impl ==> add
better?: (K, K) -> Boolean

union(l1, l2) == setUnion(l1, l2)
varselect(l, x) == [k for k in l | member?(x, variables(k::F))]
ksec(k, l, x) == kmax setUnion(remove(k, l), vark(argument k, x))

vark(l, x) ==
  varselect(reduce("setUnion",[kernels f for f in l],empty()$List(K)), x)

kmax l ==
  ans := first l
  for k in rest l repeat
    if better?(k, ans) then ans := k
  ans

-- true if x should be considered before y in the tower
better?(x, y) ==
  height(y) ^= height(x) => height(y) < height(x)
  has?(operator y, ALGOP) or
  (is?(y, "exp":SE) and not is?(x, "exp":SE)
   and not has?(operator x, ALGOP))

if R has IntegralDomain then
  removeConstantTerm(f, x) ==
    not freeOf?((den := denom f)::F, x) => f
    (u := isPlus(num := numer f)) case "failed" =>
      freeOf?(num::F, x) => 0
      f
    ans:P := 0
    for term in u::List(P) repeat
      if not freeOf?(term::F, x) then ans := ans + term
    ans / den
if \( R \) has GcdDomain and \( F \) has ElementaryFunctionCategory then

\[
\begin{align*}
\text{psimp} & : (P, SE) \rightarrow \text{Record}(\text{coef}: \text{Integer}, \text{logand}: F) \\
\text{cont} & : (P, \text{List } K) \rightarrow P \\
\text{logsimp} & : (F, SE) \rightarrow F \\
\text{linearLog?} & : (K, F, SE) \rightarrow \text{Boolean}
\end{align*}
\]

\[
\text{logsimp}(f, x) == \\
r1 := \text{psimp}(\text{numer } f, x) \\
r2 := \text{psimp}(\text{denom } f, x) \\
g := \gcd(r1.\text{coef}, r2.\text{coef}) \\
g * \log(r1.\text{logand}^{(r1.\text{coef} \div g) / r2.\text{logand}^{(r2.\text{coef} \div g)}})
\]

\[
\text{cont}(p, l) == \\
\text{empty? } l \Rightarrow p \\
q := \text{univariate}(p, \text{first } l) \\
\text{cont}(\text{unitNormal}(\text{leadingCoefficient } q).\text{unit} \ast \text{content } q, \text{rest } l)
\]

\[
\text{linearLog?}(k, f, x) == \\
\text{is?}(k, \text{"log"}: SE) \text{ and } \\
((u := \text{retractIfCan}(\text{univariate}(f,k)@\text{Union}(\text{UP},\text{"failed"})) \text{ case } \text{UP}) \\
\text{ and } \text{one?}(\text{degree}(u::\text{UP})) \\
\text{ and } (\text{degree}(u::\text{UP}) = 1) \\
\text{ and } \text{not member?}(x, \text{variables leadingCoefficient}(u::\text{UP}))
\]

\[
\text{mkPrim}(f, x) == \\
lg := [k \text{ for } k \text{ in kernels } f | \text{linearLog?}(k, f, x)] \\
\text{eval}(f, lg, [\text{logsimp(first argument } k, x) \text{ for } k \text{ in } lg])
\]

\[
\text{psimp}(p, x) == \\
(u := \text{isExpt}(p := ((p \text{ exquo } \text{cont}(p, \text{varexpt} \text{variables } p, x)))::P))) \\
\text{case } \text{"failed"} \Rightarrow [1, p::F] \\
[u.\text{exponent}, u.\text{var}::F]
\]

if \( R \) has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer) and \( F \) has Join(LiouvilleFunctionCategory, RetractableTo SE) then

\[
\begin{align*}
\text{intPatternMatch}(f, x, \text{int, pmint}) & == \\
\text{ir} & := \text{int}(f, x) \\
\text{empty?}(l := \text{nolem} \text{ir}) & \Rightarrow \text{ir} \\
\text{ans} & := \text{ratpart} \text{ir} \\
\text{nl} & := \text{List(Record(integrand: } F, \text{intvar: } F)) \Rightarrow \text{empty()} \\
\text{lg} & := \text{logpart} \text{ir} \\
\text{for } \text{rec in } l \text{ repeat} \\
\text{u} & := \text{pmint}(\text{rec}.\text{integrand}, \text{retract(}\text{rec}.\text{intvar})) \\
\text{if } u \text{ case } \text{ANS} \text{ then} \\
\text{rc} & := u::\text{ANS} \\
\text{ans} & := \text{ans} \ast \text{rc}.\text{special} \\
\text{if } \text{rc}.\text{integrand} ^= 0 \text{ then} \\
\text{ir0} & := \text{intPatternMatch}(\text{rc}.\text{integrand}, x, \text{int, pmint}) \\
\text{ans} & := \text{ans} \ast \text{ratpart} \text{ir0}
\end{align*}
\]
package IPRNTPK InternalPrintPackage

---

InternalPrintPackage.input

)set break resume
)sys rm -f InternalPrintPackage.output
)spool InternalPrintPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InternalPrintPackage
--E 1

)spool
)lisp (bye)

---

InternalPrintPackage.help

====================================================================
InternalPrintPackage examples
====================================================================

A package to print strings without line-feed nor carriage-return.
See Also:
* show InternalPrintPackage

---

**InternalPrintPackage (IPRNTPK)**

Exports:
- `iprint`:

--- package IPRNTPK InternalPrintPackage ---

```lisp
)abbrev package IPRNTPK InternalPrintPackage
++ Author: Themos Tsikas
++ Date Created: 09/09/1998
++ Date Last Updated: 09/09/1998
++ Description:
++ A package to print strings without line-feed nor carriage-return.

InternalPrintPackage(): Exports == Implementation where

Exports == with
  iprint: String -> Void
  ++ \axiom{iprint(s)} prints \axiom{s} at the current position
  ++ of the cursor.

Implementation == add
  iprint(s: String) ==
    PRINC(coerce(s)@Symbol)$Lisp
    FORCE_-OUTPUT()$Lisp
```

---
package IRURPK InternalRationalUnivariateRepresentationPackage

--- InternalRationalUnivariateRepresentationPackage.input ---

)set break resume
)sys rm -f InternalRationalUnivariateRepresentationPackage.output
)spool InternalRationalUnivariateRepresentationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InternalRationalUnivariateRepresentationPackage
--E 1

)spool
)lisp (bye)

---

--- InternalRationalUnivariateRepresentationPackage.help ---

====================================================================
InternalRationalUnivariateRepresentationPackage examples
====================================================================

An internal package for computing the rational univariate representation of a zero-dimensional algebraic variety given by a square-free triangular set. The main operation is rur.

See Also:
  o )show InternalRationalUnivariateRepresentationPackage
InternalRationalUnivariateRepresentationPackage (IRURPK)

Exports:
checkRur rur

— package IRURPK InternalRationalUnivariateRepresentationPackage —

)abbrev package IRURPK InternalRationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ References:
++ Description:

InternalRationalUnivariateRepresentationPackage(R,E,V,P,TS): Exports == Implementation where
R : Join(EuclideanDomain,CharacteristicZero)
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LV ==> List V
LP ==> List P
PWT ==> Record(val: P, tower: TS)
LPWT ==> Record(val: LP, tower: TS)
WIP ==> Record(pol: P, gap: Z, tower: TS)
BWT ==> Record(val:Boolean, tower: TS)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
normpack ==> NormalizationPackage(R,E,V,P,TS)

Exports == with

rur: (TS,B) -> List TS
  ++ \spad{rur(ts,univ?)} returns a rational univariate representation
  ++ of \spad{ts}. This assumes that the lowest polynomial in \spad{ts}
  ++ is a variable \spad{v} which does not occur in the other polynomials
  ++ of \spad{ts}. This variable will be used to define the simple
  ++ algebraic extension over which these other polynomials will be
  ++ rewritten as univariate polynomials with degree one.
  ++ If \spad{univ?} is \spad{true} then these polynomials will have
  ++ a constant initial.
checkRur: (TS, List TS) -> Boolean
  ++ \spad{checkRur(ts,lus)} returns \spad{true} if \spad{lus}
  ++ is a rational univariate representation of \spad{ts}.

Implementation == add

checkRur(ts: TS, lts: List TS): Boolean ==
  f0 := last(ts)::P
  z := mvar(f0)
  ts := collectUpper(ts,z)
  dts: N := degree(ts)
  lp := parts(ts)
  dlts: N := 0
  for us in lts repeat
    dlts := dlts + degree(us)
    rem := [removeZero(p,us) for p in lp]
    not every?(zero?,rem) =>$\endgroup$
    output(us::OutputForm)$\begingroup$OutputPackage
    return false
  if dts =$N dlts$@Boolean
convert(p:P,sqfr?:B):TS ==
  -- if sqfr? ASSUME p is square-free
  newts: TS := empty()
  sqfr? => internalAugment(p,newts)
  p := squareFreePart(p)
  internalAugment(p,newts)
prepareRur(ts: TS): List LPWT ==
  not purelyAlgebraic?(ts)$TS =>$\endgroup$
  error "$\期末\text{rur}\$/\text{IRURPK}: \#1 is not zero-dimensional"
  lp: LP := parts(ts)$TS
  lp := sort(infrittw?,lp)
  empty? lp =>$\endgroup$
  error "$\期末\text{rur}\$/\text{IRURPK}: \#1 is empty"
  f0 := first lp; lp := rest lp
CHAPTER 10. CHAPTER I

-- not (one?(init(f0)) and one?(mdeg(f0)) and zero?(tail(f0))) =>
not ((init(f0) = 1) and (mdeg(f0) = 1) and zero?(tail(f0))) =>
error "rur$IRURPK: #1 has no generating root."
empty? lp =>
error "rur$IRURPK: #1 has a generating root but no indeterminates"
z: V := mvar(f0)
f1 := first lp; lp := rest lp
x1: V := mvar(f1)
newf1 := x1::P - z::P
toSave: List LPWT := []
for ff1 in irreducibleFactors([f1])$polsetpack repeat
  newf0 := eval(ff1,mvar(f1),f0)
  ts := internalAugment(newf1,convert(newf0,true)@TS)
toSave := cons([lp,ts],toSave)
toSave

--ASSUME r is a irreducible univariate polynomial in z
--ASSUME c and s only depends on z and mvar(s)
--ASSUME c and a have main degree 1
--ASSUME c and s have a constant initial
--ASSUME mvar(ts) < mvar(s)
lp: LP := parts(ts)
lp := sort(infRittWu?,lp)
newts: TS := convert(r,true)@TS
s := remainder(s,newts).polnum
if univ?
  then
    s := normalizedAssociate(s,newts)$normpack
for p in lp repeat
  p := lazyPrem(eval(p,z,c),s)
  p := remainder(p,newts).polnum
  newts := internalAugment(p,newts)
  internalAugment(s,newts)

next(lambda:Z):Z ==
  if lambda < 0 then lambda := - lambda + 1 else lambda := - lambda

-- if check? THEN some VERIFICATIONS are performed
-- if info? THEN some INFORMATION is displayed
f0 := last(ts)::P
z: V := mvar(f0)
lambda: Z := 1
ts := collectUpper(ts,z)
toSee: List WIP := [[f0,lambda,ts]$WIP]
toSave: List TS := []
while not empty? toSee repeat
  wip := first toSee; toSee := rest toSee
  (f0, lambda, ts) := (wip.pol, wip.gap, wip.tower)
if check? and ((not univariate?(f0)$polsetpack) or (mvar(f0) ^= z))
then
    output("Bad f0: ")$OutputPackage
    output(f0::OutputForm)$OutputPackage
    c: P := lambda * xi::P + z::P
    f := eval(f0,z,c); q := eval(p,z,c)
    prs := subResultantChain(q,f)
    r := first prs; prs := rest prs
    check? and ((not zero? degree(r,xi)) or (empty? prs)) =>
        error "rur$IRURPK: should never happen !"
    s := first prs; prs := rest prs
    check? and (zero? degree(s,xi)) and (empty? prs) =>
        error "rur$IRURPK: should never happen !!"
    if zero? degree(s,xi) then s := first prs
    -- not one? degree(s,xi) =>
    not (degree(s,xi) = 1) =>
        toSee := cons([f0,next(lambda),ts]$WIP,toSee)
        h := init(s)
        r := squareFreePart(r)
        ground?(h) or ground?(gcd(h,r)) =>
            for fr in irreducibleFactors([r])$polsetpack repeat
                ground? fr => "leave"
                toSave := cons(makeMonic(z,c,fr,ts,s,univ?),toSave)
if info?
then
    output("Unlucky lambda")$OutputPackage
    output(h::OutputForm)$OutputPackage
    output(r::OutputForm)$OutputPackage
    toSee := cons([f0,next(lambda),ts]$WIP,toSee)
toSave

rur (ts: TS,univ?:Boolean): List TS ==
toSee: List LPWT := prepareRur(ts)
toSave: List TS := []
while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    ts: TS := wip.tower
    lp: LP := wip.val
    empty? lp => toSave := cons(ts,toSave)
    p := first lp; lp := rest lp
    xi: V := mvar(p)
    p := remainder(p,ts).polnum
    if not univ?
then
    p := primitivePart stronglyReduce(p,ts)
    ground?(p) or (mvar(p) < xi) =>
        error "rur$IRURPK: should never happen"
    -- (one? mdeg(p)) and (ground? init(p)) =>
    (mdeg(p) = 1) and (ground? init(p)) =>
    ts := internalAugment(p,ts)
\[
\begin{align*}
\text{wip} & := [lp, ts] \\
\text{toSee} & := \text{cons}(\text{wip}, \text{toSee}) \\
\text{lts} & := \text{makeLinearAndMonic}(p, xi, ts, \text{univ?}, \text{false}, \text{false}) \\
\text{for ts in lts repeat} & \\
\text{wip} & := [lp, ts] \\
\text{toSee} & := \text{cons}(\text{wip}, \text{toSee}) \\
\text{toSave} & \\
\end{align*}
\]

——

IRURPK.dotabb ——

"IRURPK" [color="#FF4488", href="bookvol10.4.pdf#nameddest=IRURPK"]
"SFRTCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"IRURPK" -> "SFRTCAT"

——

package INTFRSP InterpolateFormsPackage

—— InterpolateFormsPackage.input ——

)set break resume
)sys rm -f InterpolateFormsPackage.output
)spool InterpolateFormsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterpolateFormsPackage

--R
--R InterpolateFormsPackage(K: Field, symb: List(Symbol), PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)), E: ... LocalPowerSeriesCategory(K), Plc: PlacesCategory(K,PCS), DIVISOR: DivisorCategory(Plc)) is a package constructor
--R Abbreviation for InterpolateFormsPackage is INTFRSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTFRSP
--R
--R--------------------------------------------- Operations ---------------------------------------------
--R basisOfInterpolateForms : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R basisOfInterpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R interpolateForms : (DIVISOR,NonNegativeInteger,PolyRing,List(PolyRing)) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R
--E 1
InterpolateFormsPackage (INTFRSP)

--- package INTFRSP InterpolateFormsPackage ---

)abbrev package INTFRSP InterpolateFormsPackage
++ Author: Gaétan Hache
++ Date Created: 17 Nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
InterpolateFormsPackage(K,symb,PolRing,E,ProjPt,PCS,Plc,DIVISOR):_
Exports == Implementation where
  K:Field
  symb: List(Symbol)
  OV ==> OrderedVariableList(symb)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  PolRing : PolynomialCategory(K,E,OV)
  ProjPt : ProjectiveSpaceCategory(K)
  PCS : LocalPowerSeriesCategory(K)
  Plc : PlacesCategory(K,PCS)
  DIVISOR : DivisorCategory(Plc)
  INT ==> Integer
  NNI ==> NonNegativeInteger
  ParamPack ==> ParametrizationPackage(K,symb,PolRing,E,ProjPt,PCS,Plc)
  PackPoly ==> PackageForPoly(K,PolRing,E,#symb)
  LINPACK ==> LinearSystemFromPowerSeriesPackage(K,PCS)

Exports ==> with
  basisOfInterpolateForms: (DIVISOR,List PolRing) -> List(Vector(K))
  basisOfInterpolateFormsForFact: (DIVISOR,List PolRing) -> List(Vector(K))
  interpolateFormsForFact: (DIVISOR,List PolRing) -> List(PolRing)
  interpolateForms: (DIVISOR,NNI,PolRing,List PolRing) -> List(PolRing)
    interpolateForms(D,n,pol,base) compute the basis of the sub-vector
    space \( W \) of \( V = \langle \text{base} \rangle \), such that for all \( G \) in \( W \), the
    divisor \( (G) \geq D \). All the elements in base must be homogeneous
    polynomial of degree \( n \). Typically, base is the set of all monomial
    of degree \( n \); in that case, interpolateForms(D,n,pol,base)
    returns the basis of the vector space of all forms of degree \( d \) that
    interpolated \( D \). The argument \( \text{pol} \) must be the same polynomial that
    defined the curve form which the divisor \( D \) is defined.

Implementation ==> add
  import PolRing
  import PCS

  sbSpcOfCurve: (NNI,PolRing) -> List(List(K))
  exponent2monomial: List(NNI) -> PolRing
  crtV: (List(K),List(INT),NNI) -> List(K)
  createLinSys: (List Plc, List INT,List PolRing) -> Matrix(K)
CREATE "INTERPOLATEFORMSPACKAGE"

createLinSysWOVectorise: (List Plc, List INT,List PolyRing) -> Matrix(K)

basisOfInterpolateFormsForFact(divis,lm)==
  -- permet d'interpoler un diviseur qui n'est pas rationnel.
  -- La partie non rationel
  -- est dans sptdiv (note: une place de sptdiv est une place qui identifie
  -- l'ensemble des places qui lui sont conjuguees.
  -- Note: On utilise ici la fonction createLinSysWOVectorise
  -- qui ne vectorise pas les elements du corps de base.
  lstOfPlc:= supp divis
  lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
  -- ppsol contiendra la base des formes interpolant ke diviseur divis
  linSys:Matrix(K)
  linSysT:Matrix(K)
  ll:List Matrix K
  "empty?(lstOfPlc) =>
  linSys:=createLinSysWOVectorise(lstOfPlc,lstOfv,lm)
  nullSpace linSys
  zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
  nullSpace zeroMat

interpolateForms(divis,d,laCrb,lm)==
  -- ppsol contiendra la base des formes interpolant le diviseur divis
  -- mieux vaut prendre divOfZero de divis ?
  ppsol:= basisOfInterpolateForms(divis,lm)
  ppsol:List(List(K)):= [entries(vec) for vec in ppsol]
  mpsol:=psol
  sbspc:List(List(K))
  if ~(totalDegree(laCrb)$PackPoly > d) then
    -- retourne une base des formes de degres d
    sbspc:=sbSpcOfCurve(d,laCrb)
    mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
  empty?(mpsol) => [0]
  rowEchmpsol:=rowEchelon(matrix(mpsol)$Matrix(K))
  npsol:=listOfLists(rowEchmpsol)
  [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]

interpolateFormsForFact(divis,lm)==
  -- ppsol contiendra la base des formes interpolant le diviseur divis
  ppsol:= basisOfInterpolateFormsForFact(divis,lm)
  ppsol:List(List(K)):= [entries(vec) for vec in ppsol]
  mpsol:=psol
  empty?(mpsol) => [0]
  rowEchmpsol:=rowEchelon matrix(mpsol)$Matrix(K)
  npsol:=listOfLists(rowEchmpsol)
[reduce("+", [a*f for a in ll for f in lm]) for ll in npsol]

createLinSys(lstOfPlc, lstOfv, lm) ==
  lplsT:=[ [parametrize(f,pl)$ParamPack for f in lm] _
     for pl in lstOfPlc]
  lpls:=[[filterUpTo(s,v) for s in souslplsT] _
         for souslplsT in lplsT _
     for v in lstOfv]
  linSys:=reduce("vertConcat",_
     [finiteSeries2LinSys(souslplsT,v)$LINPACK_ _
      for souslplsT in lpls _
    for v in lstOfv])
  linSys

createLinSysWOVectorise(lstOfPlc, lstOfv, lm) ==
  lplsT:=[ [parametrize(f,pl)$ParamPack for f in lm] _
     for pl in lstOfPlc]
  lpls:=[[filterUpTo(s,v) for s in souslplsT] _
         for souslplsT in lplsT _
     for v in lstOfv]
  linSys:=reduce("vertConcat",_
     [finiteSeries2LinSysWOVectorise(souslplsT,v)$LINPACK_ _
      for souslplsT in lpls _
    for v in lstOfv])
  linSys

basisOfInterpolateForms(divis, lm) ==
  lstOfPlc:= supp divis
  lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
  -- ppsol contiendra la base des formes interpolant ke diviseur divis
  linSys:=Matrix(K)
  ~empty?(lstOfPlc) =>
    linSys:=createLinSys(lstOfPlc, lstOfv, lm)
  -- ppsol contiendra la base des formes passant par le diviseur divv
  nullSpace(linSys)
  zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
  nullSpace zeroMat

-- interpolateForms(divis,d,laCrb,lm)==
-- lstOfPlc:= supp divis
-- lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
--
-- lpls : List(List(PCS))
-- lplsT: List(List(PCS))
--
-- -- ppsol contiendra la base des formes interpolant ke diviseur divis
-- ppsol:List(Vector(K))
-- linSys:Matrix(K)
-- if ~empty?(lstOfPlc) then
-- linSys:=createLinSys(lstOfPlc,lstOfv,lm)
--
-- -- ppsol contient la base des formes passant par le diviseur divv
-- ppsol:=nullSpace(linSys)
--
-- else
-- zeroMat:=Matrix(K):=zero(1,#lm)$Matrix(K)
--
-- mpsol:=psol:List(List(K)):=[entries(vec) for vec in ppsol]
--
-- if ^(totalDegree(laCrb) > d) then
--
-- -- retourne une base des formes de degres d
--
-- -- qui sont un multiple de la courbe
--
-- sbspc:=sbSpcOfCurve(d,laCrb)
--
-- mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
--
-- empty?(mpsol) => [0]
--
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
--
-- npsol:=listOfLists(rowEchmpsol)
--
-- [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]
--
-- interpolateForms(divis,d,laCrb,lm)==
-- lstOfPlc:= supp divis
--
-- lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
--
-- lpls : List(List(PCS))
-- lplsT: List(List(PCS))
--
-- -- ppsol contiendra la base des formes interpolant ke diviseur divis
--
-- ppsol:List(Vector(K))
--
-- linSys:Matrix(K)
--
-- if ^empty?(lstOfPlc) then
--
-- lplsT:=[ [parametrize(f,pl)$ParamPack for f in lm]
-- for pl in lstOfPlc]
--
--
-- lpls:=[[filterUpTo(s,v) for s in souslplsT] _
-- for souslplsT in lplsT_ _
-- for v in lstOfv]
--
-- linSys:=reduce("vertConcat",_
-- [finiteSeries2LinSys(souslplsT,v)$LINPACK_ 
-- for souslplsT in lpls_ _
-- for v in lstOfv])
--
--
-- -- ppsol contient la base des formes passant par le diviseur divv
--
-- ppsol:=nullSpace(linSys)
--
-- else
--
-- zeroMat:=Matrix(K):=zero(1,#lm)$Matrix(K)
-- ppsol:=nullSpace zeroMat
-- mpsol:=psol:List(List(K)):=\{entries(vec) for vec in ppsol\}
--
-- if \((totalDegree(laCrb) > d)\) then
--  -- retourne une base des formes de degr\'{e}s d
--  -- qui sont un multiple de la courbe
--  sbspc:=sbSpcOfCurve(d,laCrb)
--  mpsol:=quotVecSpaceBasis(psol,sbspc)\$_{LinesOpPack(K)}$
--
-- empty?(mpsol) => [0]
--
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
-- npsol:=listOfLists(rowEchmpsol)
-- [reduce("+",\{a*f for a in ll for f in lm\}) for ll in npsol]

listVar:List(OV):= \{index(i::PositiveInteger)\$_{OV}$ for i in 1..#symb\}

listMonoPols:List(PolyRing):=\{monomial(1,vv,1) for vv in listVar\}

crtV(lcoef,lpos,1)==
  vvv:List(K):=\{0 for i in 1..1\}
  for c in lcoef for p in lpos repeat
    setelt(vvv,p,c)
  vvv

sbSpcOfCurve(m,laCrb)==
  d:=totalDegree(laCrb)\$_{PackPoly}$
  lm:List(PolyRing):=listAllMono(m)\$_{PackPoly}$
  m<d => [[0$K$ for i in 1..#lm]]
  sd:NNI:=((m pretend INT)-(d pretend INT)) pretend NNI
  slm:List(PolyRing):=listAllMono(sd)\$_{PackPoly}$
  allPol:=\{laCrb*f for f in slm\}
  lpos:=\{\{position(m,lm) for m in primitiveMonomials(f)\} for f in allPol\}
  lcoef:=\{coefficients(f) for f in allPol\}
  clm:=#lm
  [crtV(lc,lp,clm) for lc in lcoef for lp in lpos]

inVecSpace?: (List(K),List(List(K))) -> Boolean
inVecSpace?(line,basis)==
  mat:=Matrix(K):=matrix(basis)
  rmat:=rank(mat)
  augmat:=Matrix(K):=matrix(concat(line,basis))
  raugmat:=rank(augmat)
  rmat=raugmat

exponent2monomial(lexp)==
  reduce("*",\{m**e for m in listMonoPols for e in lexp\})

-- interpolateFunctions(lstOfPlc,lstOfv,lmnumer)==
-- lstOfPlc:= supp divis
-- lstOfv:= [coef(divis,pl) for pl in lstOfPlc]
--
-- lpls:List(List(PCS))
-- lplsT:List(List(PCS))
-- llll:List(List(Integer))
-- 10rd:List(Integer)
-- ordMin:Integer
-- ppsol:List(Vector(K))
-- linSys:Matrix(K)
-- if "empty?(lstOfPlc) then
-- lplsT:=[[parametrize(f,pl)$ParamPack for f in lmnumer ] _
-- for pl in lstOfPlc]
-- lplsT:=[[removeFirstZeroes(s) for s in l] for l in lplsT]
--
-- -- series must be shift if summe of them has negative order
-- llll:= [[order(s)$PCS for s in l] for l in lplsT]
-- 10rd:= concat llll
-- 10rd:=cons(0,10rd)
-- ordMin:Integer:= "min"/10rd
-- lplsT:=[[shift(s,-ordMin) for s in l] for l in lplsT]
--
-- lpls:=[[filterUpTo(s,v-ordMin) for s in souslplsT] _
-- for souslplsT in lplsT for v in lstOfv]
-- linSys:=reduce("vertConcat",_
-- [finiteSeries2LinSys(souslplsT,v-ordMin)$LINPACK _
-- for souslplsT in lpls for v in lstOfv])
--
-- ppsol contient la base des formes passant par le diviseur divv
-- ppsol:=nullSpace(linSys)
-- else
-- zeroMat:Matrix(K):=zeros(1,#lmnumer)$Matrix(K)
-- ppsol:=nullSpace zeroMat
-- mpsol:=psol:List(List(K)):=[entries(vec) for vec in ppsol]
-- -- inserer ici le code pour calculer la base modulo l'ideal ...
-- empty?(mpsol) => [0]
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
-- npsol:=listOfLists(rowEchmpsol)
-- [reduce("+",[a*f for a in ll for f in lmnumer]) for ll in npsol]
package INTDIVP IntersectionDivisorPackage

--- IntersectionDivisorPackage.input ---

)set break resume
)sys rm -f IntersectionDivisorPackage.output
)spool IntersectionDivisorPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntersectionDivisorPackage
--R
--R IntersectionDivisorPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,Ordering),Polys: List(Polynomial(K,E)),DesingTreeCategory(InfClsPoint),BLMET: BlowUpMethodCategory) is a package constructor
--R Abbreviation for IntersectionDivisorPackage is INTDIVP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTDIVP
--R
--R------------------------------------ Operations ---------------------------------
--R intersectionDivisor : (PolyRing,PolyRing,List(DesTree),List(ProjPt)) -> DIVISOR
--R placesOfDegree : (PositiveInteger,PolyRing,List(ProjPt)) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- IntersectionDivisorPackage.help ---

====================================================================
IntersectionDivisorPackage examples
====================================================================

The following is part of the PAFF package

See Also:
  o )show IntersectionDivisorPackage
IntersectionDivisorPackage (INTDIVP)

Exports:
  intersectionDivisor  placesOfDegree

— package INTDIVP IntersectionDivisorPackage —

)abbrev package INTDIVP IntersectionDivisorPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt, PCS,Plc,DIVISOR, _ InfClsPoint,DesTree,BLMET):

Exports == Implementation where

K:Field
symb: List(Symbol)

OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)

PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)
InfClsPoint : InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt, _ PCS,Plc,DIVISOR,BLMET)

DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory
OF => OutputForm
PackPoly => PackageForPoly(K,PolyRing,E,#symb)
PPFC1 => PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
ParamPackFC => LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_,E,ProjPt,PCS,Plc)
ParamPack => ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
RatSingPack => ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
DesingPack => DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,_,DIVISOR,InfClsPoint,DesTree,BLMET)

Exports => with

intersectionDivisor:(PolyRing,PolyRing,List DesTree,List ProjPt) -> DIVISOR
++ intersectionDivisor(f,pol,listOfTree) returns the intersection
++ divisor of f with a curve defined by pol. listOfTree must contain
++ all the desingularisation trees of all singular points on the curve
++ defined by pol.

placesOfDegree: (PositiveInteger, PolyRing, List ProjPt) -> Void()
++ placesOfDegree(d, f, pts) compute the places of degree
++ dividing d of the curve f. pts should be the singular points
++ of the curve f. For d > 1 this only works if K has
++ \axiomType{PseudoAlgebraicClosureOfFiniteFieldCategory}.

Implementation => add

intersectionDivisor(pol,curve,ltr,listOfSingPt)==
    intDeg:Integer:= (totalDegree(pol)$PackPoly * _
                        totalDegree(curve)$PackPoly) pretend Integer
    -- compute at places over singular Points
    lDivAtSingPt:DIVISOR:=_
        reduce("+",[divisorAtDesingTree(pol,tr)$DesingPack for tr in ltr],0)
    -- By Bezout Thorem, if all intersection points with mult.
    -- have been found then return the divisor
    degD:Integer:=degree lDivAtSingPt
    degD = intDeg => lDivAtSingPt
    setOfFdPlc:List Plc:=[pl for pl in setOfFdPlc | "leaf?(pl)]
    plcFrSplPts:List Plc:=[pl for pl in setOfFdPlc | "leaf?(pl)]
    ordAtPlcFrSplPts:List Integer:=_
        [order(parametrize(pol,pl)$ParamPack)$PCS for pl in plcFrSplPts]
    divAtSplPts:DIVISOR:=_
        reduce("*",[o * (pl :: DIVISOR) _
                        for o in ordAtPlcFrSplPts,pl in plcFrSplPts],0)
    tDiv:=lDivAtSingPt+divAtSplPts
    -- By Bezout Thorem, if all intersection points with mult.
    -- have been found then return the divisor
    degD:Integer:=degree tDiv
    degD = intDeg => tDiv
intPts: List ProjPt := algebraicSet([pol, curve])$RatSingPack
intPtsNotSing := setDifference(intPts, listOfSingPt)
ipPtsList := List(Plc) :=
    [pointToPlace(pt, curve)$ParamPackFC for pt in intPtsNotSing]
remPlc := setDifference(intPtsList, plcFrSplPts)
ordAtPlcRemList :=
    [order(parametrize(pol, pl)$ParamPack)$PCS for pl in remPlc]
divAtRemDivisor :=
    reduce(\("\), [\(\(o \cdot (pl :: \text{DIVISOR})\) for o in ordAtPlcRemList, pl in remPlc], 0)
theDivisor := lDivAtSingPt + divAtSplPts + divAtRem
degD := degree theDivisor
if \(\text{degD} = \text{intDeg}\) then
    print("error while computing the intersection divisor" :: OF )
    print("Otherwise the Bezout Theoreme is not true ! ! ! !" :: OF)
print("Of course its the machine that make the mistake ! ! ! ! !") :: OF)
theDivisor

placesOfDegree(d, curve, singPts) :=
    --Return the number of places of degree \(i\) of the functionfield, no
    --constant field extension
    allPoints := List ProjPt := rationalPoints(curve, d)$RatSingPack
    remindingSimplePts := List ProjPt := setDifference(allPoints, singPts)
    for tpt in remindingSimplePts repeat
        pointToPlace(tpt, curve)$ParamPackFC
        Void()

package IRREDFFX IrredPolyOverFiniteField

    — IrredPolyOverFiniteField.input —

)set break resume
)sys rm -f IrredPolyOverFiniteField.output
)spool IrredPolyOverFiniteField.output
)set message test on
This package exports the function generateIrredPoly that computes a monic irreducible polynomial of degree n over a finite field.

See Also:
- \texttt{\textbackslash show IrredPolyOverFiniteField}

---

\textbf{IrredPolyOverFiniteField (IRREDFFX)}

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{irredffx.png}
\caption{Diagram of \texttt{IrredPolyOverFiniteField} package}
\end{figure}

\textbf{Exports:}
- \texttt{generateIrredPoly}

---

\textbf{--- package IRREDFFX IrredPolyOverFiniteField ---}
++)abbrev package IRREDFFX IrredPolyOverFiniteField
++ Author: Robert S. Sutor (original)
++ Date Last Updated: 29 May 1990
++ Description:
++ This package exports the function generateIrredPoly that computes
++ a monic irreducible polynomial of degree n over a finite field.

IrredPolyOverFiniteField(GF:FiniteFieldCategory): Exports == Impl where
  N ==> PositiveInteger
  Z ==> Integer
  SUP ==> SparseUnivariatePolynomial GF
  QR ==> Record(quotient: Z, remainder: Z)

Exports ==> with
  generateIrredPoly: N -> SUP
    ++ generateIrredPoly(n) generates an irreducible univariate
    ++ polynomial of the given degree n over the finite field.

Impl ==> add
  import DistinctDegreeFactorize(GF, SUP)

getIrredPoly : (Z, N) -> SUP
getIrredPoly(start : Z, n : N) : SUP ==
  -- idea is to iterate over possibly irreducible monic polynomials
  -- until we find an irreducible one. The obviously reducible ones
  -- are avoided.
  mon := monomial(1, n)$SUP
  pol: SUP := 0
  found: Boolean := false
  end: Z := q**n - 1
  while not ((end < start) or found) repeat
    if gcd(start, p) = 1 then
      if irreducible?(pol := mon + qAdicExpansion(start)) then
        found := true
      end:
    start := start + 1
  zero? pol => error "no irreducible poly found"
generateIrredPoly(n : N) : SUP ==
-- want same poly every time
-- one?(n) => monomial(1, 1)$SUP
(n = 1) => monomial(1, 1)$SUP
-- one?(gcd(p, n)) or (n < q) =>
(gcd(p, n) = 1) or (n < q) =>
odd?(n) => getIrredPoly(2, n)
getIrredPoly(1, n)
getIrredPoly(q + 1, n)

package IRSN IrrRepSymNatPackage

— IrrRepSymNatPackage.input —

)set break resume
)sys rm -f IrrRepSymNatPackage.output
)spool IrrRepSymNatPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show IrrRepSymNatPackage
--E 1

)spool
)lisp (bye)

— IrrRepSymNatPackage.help —
IrrRepSymNatPackage examples

IrrRepSymNatPackage contains functions for computing
the ordinary irreducible representations of symmetric groups on
n letters \{1,2,...,n\} in Young’s natural form and their dimensions.

These representations can be labelled by number partitions of n,
i.e. a weakly decreasing sequence of integers summing up to n, e.g.

Note that whenever a \spadtype{List Integer} appears in a signature,
a partition required.

See Also:
o \show IrrRepSymNatPackage

---

IrrRepSymNatPackage (IRSN)

Exports:
  dimensionOfIrreducibleRepresentation  irreducibleRepresentation

--- package IRSN IrrRepSymNatPackage ---

)abbrev package IRSN IrrRepSymNatPackage
++ Authors: Johannes Grabmeier, Thorsten Werther
++ Date Created: 04 August 1988
++ Date Last Updated: 24 May 1991
++ References:
++ G. James, A. Kerber: The Representation Theory of the Symmetric
++ J. Grabmeier, A. Kerber: The Evaluation of Irreducible
++ Polynomial Representations of the General Linear Groups
++ and of the Unitary Groups over Fields of Characteristic 0,
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
++ Description:
++ IrrRepSymNatPackage contains functions for computing
++ the ordinary irreducible representations of symmetric groups on
++ n letters \{1,2,...,n\} in Young’s natural form and their dimensions.
++ These representations can be labelled by number partitions of n,
++ i.e. a weakly decreasing sequence of integers summing up to n, e.g.
++ \[3,3,1\] labels an irreducible representation for n equals 10.
++ Note that whenever a \spadtype{List Integer} appears in a signature,
++ a partition required.
++ NOT TRUE in current system, but should:
++ also could be an element of \spadtype{Partition}

IrrRepSymNatPackage(): public == private where
  NNI ==> NonNegativeInteger
  I ==> Integer
  L ==> List
  M ==> Matrix
  V ==> Vector
  B ==> Boolean
  SGCF ==> SymmetricGroupCombinatoricFunctions
  ICF ==> IntegerCombinatoricFunctions Integer
  PP ==> PartitionsAndPermutations
  PERM ==> Permutation

public ==> with

  dimensionOfIrreducibleRepresentation : L I -> NNI
  ++ dimensionOfIrreducibleRepresentation(lambda) is the dimension
  ++ of the ordinary irreducible representation of the symmetric group
  ++ corresponding to lambda.
  ++ Note that the Robinson-Thrall hook formula is implemented.
  irreducibleRepresentation : (L I, PERM I) -> M I
  ++ irreducibleRepresentation(lambda,pi) is the irreducible representation
  ++ corresponding to partition lambda in Young’s natural form of the
  ++ permutation pi in the symmetric group, whose elements permute
  ++ \{1,2,...,n\}.
  irreducibleRepresentation : L I -> L M I
  ++ irreducibleRepresentation(lambda) is the list of the two
  ++ irreducible representations corresponding to the partition lambda
  ++ in Young’s natural form for the following two generators
  ++ of the symmetric group, whose elements permute
  ++ \{1,2,...,n\}, namely (1 2) (2-cycle) and
irreducibleRepresentation : (L I, L PERM I) -> L M I
  ++ irreducibleRepresentation(lambda,listOfPerm) is the list of the
  ++ irreducible representations corresponding to lambda
  ++ in Young's natural form for the list of permutations
  ++ given by listOfPerm.

private ==> add

-- local variables
oldlambda : L I := nil$(L I)
flambda : NNI := 0  -- dimension of the irreducible repr.
younglist : L M I := nil$(L M I)  -- list of all standard tableaus
lprime : L I := nil$(L I)  -- conjugated partition of lambda
n : NNI := 0  -- concerning symmetric group S_n
rows : NNI := 0  -- # of rows of standard tableau
columns : NNI := 0  -- # of columns of standard tableau
aid : M I := new(1,1,0)

-- declaration of local functions

aiIdInverse : () -> Void
  -- computes aiId, the inverse of the matrix
  -- (signum(k,l,id))_1 <= k,l <= flambda, where id
  -- denotes the identity permutation

alreadyComputed? : L I -> Void
  -- test if the last calling of an exported function concerns
  -- the same partition lambda as the previous call

listPermutation : PERM I -> L I  -- should be in Permutation
  -- converts a permutation pi into the list
  -- [pi(1),pi(2),...,pi(n)]

signum : (NNI, NNI, L I) -> I
  -- if there exists a vertical permutation v of the tableau
  -- tl := pi o younglist(1) (1-th standard tableau)
  -- and a horizontal permutation h of the tableau
  -- tk := younglist(k) (k-th standard tableau) such that
  --     v o tl = h o tk,
  -- then
  --     signum(k,l,pi) = sign(v),
  -- otherwise
  --     signum(k,l,pi) = 0.

sumPartition : L I -> NNI
  -- checks if lambda is a proper partition and results in
  -- the sum of the entries

testPermutation : L I -> NNI
-- testPermutation(pi) checks if pi is an element of S_n,
-- the set of permutations of the set \{1,2,...,n\}.
-- If not, an error message will occur, if yes it replies n.

-- definition of local functions

aIdInverse() ==
  aId := new(flambda,flambda,0)
  for k in 1..flambda repeat
    aId(k,k) := 1
  if n < 5 then return aId

  idperm : L I := nil$(L I)
  for k in n..1 by -1 repeat
    idperm := cons(k,idperm)
  for k in 1..(flambda-1) repeat
    for l in (k+1)..flambda repeat
      aId(k::NNI,l::NNI) := signum(k::NNI,l::NNI, idperm)

  return aId

-- invert the upper triangular matrix aId
  for j in flambda..2 by -1 repeat
    for i in (j-1)..1 by -1 repeat
      aId(i::NNI,j:NNI) := -aId(i::NNI,j::NNI)
    for k in (j+1)..flambda repeat
      for i in (j-1)..1 by -1 repeat
        aId(i::NNI,k:NNI) := aId(i::NNI,k::NNI) + aId(i::NNI,j::NNI) * aId(j::NNI,k::NNI)

alreadyComputed?(lambda) ==
  if not(lambda = oldlambda) then
    oldlambda := lambda
    lprime := conjugate(lambda)$PP
    rows := (first(lprime)$(L I))::NNI
    columns := (first(lambd)$L I))::NNI
    n := (+/lambda)::NNI
    younglist := listYoungTableaus(lambda)$SGCF
    flambda := #younglist
    aIdInverse() -- side effect: creates actual aId

listPermutation(pi) ==
  li : L I := nil$(L I)
  for k in n..1 by -1 repeat
    li := cons(eval(pi,k)$(PERM I),li)
  li

signum(numberOfRowTableau, numberOfColumnTableau,pi) ==
rowtab : M I := copy younglist numberOfRowTableau
columntab : M I := copy younglist numberOfColumnTableau
swap : I
sign : I := 1
end : B := false
endk : B
ctrl : B

-- k-loop for all rows of tableau rowtab
k : NNI := 1
while (k <= rows) and (not end) repeat
  -- l-loop along the k-th row of rowtab
  l : NNI := 1
  while (l <= oldlambda(k)) and (not end) repeat
    z : NNI := l
    endk := false
    -- z-loop for k-th row of rowtab beginning at column l.
    -- test whether the entry rowtab(k,z) occurs in the l-th column
    -- beginning at row k of pi o columntab
    while (z <= oldlambda(k)) and (not endk) repeat
      s : NNI := k
      ctrl := true
      while ctrl repeat
        if (s <= lprime(l))
          then
            if (1+rowtab(k,z) = pi(1+columntab(s,l)))
              -- if entries in the tableaus were from 1,...,n, then
              -- it should be ..columntab(s,l)... .
              then ctrl := false
              else s := s + 1
          else ctrl := false
        -- end of ctrl-loop
        endk := (s <= lprime(l)) -- same entry found ?
        if not endk
          then -- try next entry
            z := z + 1
        else
          if k < s
            then -- verticalpermutation
              sign := -sign
              swap := columntab(s,l)
              columntab(s,l) := columntab(k,l)
              columntab(k,l) := swap
          if l < z
            then -- horizontalpermutation
              swap := rowtab(k,z)
              rowtab(k,z) := rowtab(k,l)
              rowtab(k,l) := swap
        -- end of else
        endk := false
      -- end of inner loop
    -- end of test
  -- end of l-loop
  ctrl := true
-- end of k-loop
-- end of z-loop
if (z > oldlambda(k)) -- no corresponding entry found
then
  sign := 0
  end := true
l := l + 1
-- end of l-loop
k := k + 1
-- end of k-loop

sign

sumPartition(lambda) ==
ok : B := true
prev : I := first lambda
sum := I := 0
for x in lambda repeat
  sum := sum + x
  ok := ok and (prev >= x)
  prev := x
if not ok then
  error("No proper partition ")
sum::NNI

testPermutation(pi : L I) : NNI ==
ok : B := true
n := I := 0
for i in pi repeat
  if i > n then n := i -- find the largest entry n in pi
  if i < 1 then ok := false -- check whether there are entries < 1
-- now n should be the number of permuted objects
if (not (n=#pi)) or (not ok) then
  error("No permutation of 1,2,..,n")
-- now we know that pi has n Elements ranging from 1 to n
test : Vector(B) := new((n)::NNI,false)
for i in pi repeat
  test(i) := true -- this means that i occurs in pi
if member?(false,test) then error("No permutation") -- pi not surjective
n::NNI

-- definitions of exported functions

dimensionOfIrreducibleRepresentation(lambda) ==
bn := I := sumPartition(lambda)::I -- also checks whether lambda
dd := I := 1 -- is a partition
lambdaprime : L I := conjugate(lambda)$PP
-- run through all rows of the Youngtableau corr. to lambda
for i in 1..lambdaprime.1 repeat
  -- run through all nodes in row i of the Youngtableau
  for j in 1..lambda.i repeat
    -- the hooklength of node (i,j) of the Youngtableau
    -- is the new factor, remember counting starts with 1
    dd := dd * (lambda.i + lambdaprime.j - i - j + 1)
  
(factorial(nn)$ICF quo dd)::NNI

irreducibleRepresentation(lambda:(L I),pi:(PERM I)) ==
nn : NNI := sumPartition(lambda)
alreadyComputed?(lambda)
piList : L I := listPermutation pi
if not (nn = testPermutation(piList)) then
  error("Partition and permutation are not consistent")

aPi : M I := new(flambda,flambda,0)
for k in 1..flambda repeat
  for l in 1..flambda repeat
    aPi(k,l) := signum(k,l,piList)
  
  aId * aPi

irreducibleRepresentation(lambda) ==
listperm : L PERM I := nil$(L PERM I)
li : L I := nil$(L I)
sumPartition(lambda)
alreadyComputed?(lambda)
listperm :=
n = 1 => cons(1$(PERM I),listperm)
n = 2 => cons(cycle([1,2])$(PERM I),listperm)
-- the n-cycle (1,2,..,n) and the 2-cycle (1,2) generate S_n
for k in n..1 by -1 repeat
  li := cons(k,li) -- becomes n-cycle (1,2,..,n)
listperm := cons(cycle(li)$(PERM I),listperm)
-- 2-cycle (1,2)
cons(cycle([1,2])$(PERM I),listperm)
}

irreducibleRepresentation(lambda,listperm)
package INVLAPLA InverseLaplaceTransform

--- InverseLaplaceTransform.input ---

)set break resume
)sys rm -f InverseLaplaceTransform.output
)spool InverseLaplaceTransform.output
)set message test on
)set message auto off
)clear all

--$ 1 of 1
)show InverseLaplaceTransform
--E 1

)spool
)lisp (bye)

---

--- InverseLaplaceTransform.help ---

====================================================================
InverseLaplaceTransform examples
====================================================================

This package computes the inverse Laplace Transform.

See Also:
  o )show InverseLaplaceTransform

---
InverseLaplaceTransform (INVLAPLA)

Exports:
inverseLaplace

— package INVLAPLA InverseLaplaceTransform —

)abbrev package INVLAPLA InverseLaplaceTransform
++ Author: Barry Trager
++ Date Created: 3 Sept 1991
++ Date Last Updated: 3 Sept 1991
++ Description:
++ This package computes the inverse Laplace Transform.

InverseLaplaceTransform(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
  RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
  SpecialFunctionCategory, AlgebraicallyClosedFunctionSpace R)

SE ==> Symbol
PI ==> PositiveInteger
N ==> NonNegativeInteger
K ==> Kernel F
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP

Exports ==> with
  inverseLaplace: (F, SE, SE) -> Union(F,"failed")
  inverseLaplace(f, s, t) returns the Inverse
  Laplace transform of \spad{f(s)}
  ++ using \spad{t} as the new variable or "failed" if unable to find
  ++ a closed form.
  ++ Handles only rational \spad{f(s)}.

Implementation ==> add
-- local ops --
ilt : (F,Symbol,Symbol) -> Union(F,"failed")
ilt1 : (RF,F) -> F
iltsqfr : (RF,F) -> F
ilтирred: (UP,UP,F) -> F
freeOf?: (UP,Symbol) -> Boolean

inverseLaplace(expr,ivar,ovar) == ilt(expr,ivar,ovar)

freeOf?(p:UP,v:Symbol) ==
"and"/[freeOf?(c,v) for c in coefficients p]

ilt(expr,var,t) ==
expr = 0 => 0
r := univariate(expr,kernel(var))

-- Check that r is a rational function such that degree of
-- the numerator is lower than that degree of denominator
not(numer(r) quo denom(r) = 0) => "failed"
not( freeOf?(numer r,var) and freeOf?(denom r,var)) => "failed"
ilt1(r,t::F)

hintpac := TranscendentalHermiteIntegration(F, UP)

ilt1(r,t) ==
r = 0 => 0
rsplit := HermiteIntegrate(r, differentiate)$hintpac
-t*ilt1(rsplit.answer,t) + iltsqfr(rsplit.logpart,t)

iltsqfr(r,t) ==
r = 0 => 0
p:=numer r
q:=denom r

-- ql := [qq.factor for qq in factors factor q]
ql := [qq.factor for qq in factors squareFree q]
# ql = 1 => ilтирred(p,q,t)
# ql = 1 => ilтирred(p,q,t)

nl := multiEuclidean(ql,p)::List(UP)
+[/ilтирred(a,b,t) for a in nl for b in ql]

-- q is irreducible, monic, degree p < degree q
ilтирred(p,q,t) ==
degree q = 1 =>
cp := coefficient(p,0)
(c:=coefficient(q,0))=0 => cp
cp*exp(-c*t)
degree q = 2 =>
a := coefficient(p,1)
b := coefficient(p,0)
c:=(1/2)*coefficient(q,1)
\[
d := \text{coefficient}(q, 0) \\
e := \exp(c \cdot t) \\
b := b + a \cdot c \\
d := d - c^2 \\
d > 0 \Rightarrow \\
\alpha : \mathbb{F} := \sqrt{d} \\
e \cdot (a \cdot \cos(t \cdot \alpha) + b \cdot \sin(t \cdot \alpha)/\alpha) \\
\alpha : \mathbb{F} := \sqrt{-d} \\
e \cdot (a \cdot \cosh(t \cdot \alpha) + b \cdot \sinh(t \cdot \alpha)/\alpha) \\
\text{roots} : \text{List} \ \mathbb{F} := \text{zerosOf} \ q \\
q_1 := \text{differentiate} \ q \\
+/\{p(root)/q_1(root) \cdot \exp(root \cdot t) \text{ for root in roots}\}
\]
Chapter 11

Chapter J
package KERNEL2 KernelFunctions2

--- KernelFunctions2.input ---

)set break resume
)sys rm -f KernelFunctions2.output
)spool KernelFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show KernelFunctions2
--E 1

)spool
)lisp (bye)

---

--- KernelFunctions2.help ---

====================================================================
KernelFunctions2 examples
====================================================================

This package exports some auxiliary functions on kernels

See Also:
  o )show KernelFunctions2
KernelFunctions2 (KERNEL2)

Exports:
constantKernel constantIfCan

— package KERNEL2 KernelFunctions2 —

)abbrev package KERNEL2 KernelFunctions2
++ Description:
++ This package exports some auxiliary functions on kernels

KernelFunctions2(R:OrderedSet, S:OrderedSet): with
  constantKernel: R -> Kernel S
  ++ constantKernel(r) undocumented
  constantIfCan : Kernel S -> Union(R, "failed")
  ++ constantIfCan(k) undocumented

== add
import BasicOperatorFunctions1(R)

constantKernel r == kernel(constantOperator r, nil(), 1)
constantIfCan k == constantOpIfCan operator k

— KERNEL2.dotabb —
package KOVACIC Kovacic

— Kovacic.input —

)set break resume
)sys rm -f Kovacic.output
)spool Kovacic.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Kovacic
--E 1

)spool
)lisp (bye)

— Kovacic.help —

====================================================================
Kovacic examples
====================================================================

Kovacic provides a modified Kovacic’s algorithm for solving explicitely irreducible 2nd order linear ordinary differential equations.

See Also:
o )show Kovacic

———
Kovacic (KOVACIC)

Exports:

kovacic

— package KOVACIC Kovacic —

)abbrev package KOVACIC Kovacic
++ Author: Manuel Bronstein
++ Date Created: 14 January 1992
++ Date Last Updated: 3 February 1994
++ Description:
++ \spad{Kovacic} provides a modified Kovacic’s algorithm for
++ solving explicitly irreducible 2nd order linear ordinary
++ differential equations.

Kovacic(F, UP): Exports == Impl where
F : Join(CharacteristicZero, AlgebraicallyClosedField,
    RetractableTo Integer, RetractableTo Fraction Integer)
UP : UnivariatePolynomialCategory F
RF ==> Fraction UP
SUP ==> SparseUnivariatePolynomial RF
LF ==> List Record(factor:UP, exponent:Integer)
LODO==> LinearOrdinaryDifferentialOperator1 RF

Exports ==> with
kovacic: (RF, RF, RF) -> Union(SUP, "failed")
  ++ kovacic(a_0,a_1,a_2) returns either "failed" or P(u) such that
  ++ \spad{e^{\int(-a_1/2a_2)} e^{\int u}} is a solution of
  ++ \spad{a_2 y'' + a_1 y' + a0 y = 0}
  ++ whenever \spad{u} is a solution of \spad{P u = 0}.
  ++ The equation must be already irreducible over the rational functions.
kovacic: (RF, RF, RF, UP -> Factored UP) -> Union(SUP, "failed")
  ++ kovacic(a_0,a_1,a_2,ezfactor) returns either "failed" or P(u) such
  ++ that \spad{e^{\int(-a_1/2a_2)} e^{\int u}} is a solution of
  ++ \spad{a_2 y'' + a_1 y' + a0 y = 0}
++ whenever \spad{u} is a solution of \spad{P u = 0}.  
++ The equation must be already irreducible over the rational functions.  
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},  
++ not necessarily into irreducibles.

\texttt{Impl} \Rightarrow \texttt{add}\
\texttt{import RationalRicDE(F, UP)}

case2 : (RF, LF, UP -> Factored UP) -> Union(SUP, "failed")
cannotCase2?: LF -> Boolean
kovacic(a0, a1, a2) == kovacic(a0, a1, a2, squareFree)

-- it is assumed here that $a2 y'' + a1 y' + a0 y$ is already irreducible  
-- over the rational functions, i.e. that the associated Riccati equation  
-- does NOT have rational solutions (so we don't check case 1 of Kovacic's  
-- algorithm)  
-- currently only check case 2, not 3
kovacic(a0, a1, a2, ezfactor) ==

-- transform first the equation to the form $y'' = r y$
-- which makes the Galois group unimodular
-- this does not change irreducibility over the rational functions
-- the following is split into 5 lines in order to save a couple of  
-- hours of compile time.
r:RF := a1**2  
r := r + 2 * a2 * differentiate a1  
r := r - 2 * a1 * differentiate a2  
r := r - 4 * a0 * a2  
r := r / (4 * a2**2)  
lf := factors squareFree denom r  
case2(r, lf, ezfactor)

-- this is case 2 of Kovacic's algorithm, i.e. look for a solution  
-- of the associated Riccati equation in a quadratic extension  
-- lf is the squarefree factorisation of \texttt{denom(r)} and is used to  
-- check the necessary condition
case2(r, lf, ezfactor) ==

cannotCase2? lf => "failed"  
-- build the symmetric square of the operator $L = y'' - r y$
-- which is $L2 = y''' - 4 r y' - 2 r' y$
l2:LODO := monomial(1, 3) - monomial(4*r, 1) - 2 * differentiate(r)::LODO
-- no solution in this case if $L2$ has no rational solution
empty?(sol := ricDsolve(l2, ezfactor)) => "failed"  
-- otherwise the defining polynomial for an algebraic solution  
-- of the Ricatti equation associated with $L$ is
-- \texttt{u'2 - b u + (1/2 b' + 1/2 b'2 - r) = 0}  
-- where $b$ is a rational solution of the Ricatti of $L2$
b := first sol
monomial(1, 2)$SUP - monomial(b, 1)$SUP
    + ((differentiate(b) + b**2 - 2 * r) / (2::RF))::SUP
-- checks the necessary condition for case 2
-- returns true if case 2 cannot have solutions
-- the necessary condition is that there is either a factor with
-- exponent 2 or odd exponent > 2
cannotCase2? if ==
  for rec in lf repeat
    rec.exponent = 2 or (odd?(rec.exponent) and rec.exponent > 2) =>
      return false
  true

— KOVACIC.dotabb —

"KOVACIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=KOVACIC"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"KOVACIC" -> "ACF"

— — —
Chapter 13

Chapter L

package LAPLACE LaplaceTransform

— LaplaceTransform.input —

)set break resume
)sys rm -f LaplaceTransform.output
)spool LaplaceTransform.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LaplaceTransform
--E 1

)spool
)lisp (bye)

— LaplaceTransform.help —

====================================================================
LaplaceTransform examples
====================================================================

This package computes the forward Laplace Transform.

See Also:
  o )show LaplaceTransform

1287
LaplaceTransform (LAPLACE)

Exports:
laplace

— package LAPLACE LaplaceTransform —

)abbrev package LAPLACE LaplaceTransform
++ Author: Manuel Bronstein
++ Date Created: 30 May 1990
++ Date Last Updated: 13 December 1995
++ Description:
++ This package computes the forward Laplace Transform.

LaplaceTransform(R, F):Exports == Implementation where
R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
AlgebraicallyClosedFunctionSpace R)
SE ==> Symbol
PI ==> PositiveInteger
N ==> NonNegativeInteger
K ==> Kernel F
OFE ==> OrderedCompletion F
EQ ==> Equation OFE
ALGOP ==> "%alg"
SPECIALDIFF ==> "%specialDiff"
Exports ==> with

    laplace: (F, SE, SE) -> F
    ++ laplace(f, t, s) returns the Laplace transform of \( \text{spad}(f(t)) \)
    ++ using \( \text{spad}(s) \) as the new variable.
    ++ This is \( \text{spad}\{\text{integral}(\exp(-s*\text{t})*f(t), t = 0..\plusInfinity)) \}.
    ++ Returns the formal object \( \text{spad}(\text{laplace}(f, t, s)) \) if it cannot
    ++ compute the transform.

Implementation ==> add

    import IntegrationTools(R, F)
    import ElementaryIntegration(R, F)
    import PatternMatchIntegration(R, F)
    import PowerSeriesLimitPackage(R, F)
    import FunctionSpaceIntegration(R, F)
    import TrigonometricManipulations(R, F)

    locallaplace : (F, SE, F, SE, F) -> F
    lapkernel : (F, SE, F, F) -> Union(F, "failed")
    intlaplace : (F, F, F, SE, F) -> Union(F, "failed")
    isLinear : (F, SE) -> Union(Record(const:F, nconst:F), "failed")
    mkPlus : F -> Union(List F, "failed")
    dvlap : (List F, SE) -> F
tdenom : (F, F) -> Union(F, "failed")

algebraic? : (F, SE) -> Boolean

    oplap := operator("laplace":Symbol, 3)$BasicOperator

laplace(f,t,s) == locallaplace(complexElementary(f,t),t,t::F,s,s::F)

-- returns true if the highest kernel of \( f \) is algebraic over something
-- algebraic?(f, t) ==
    l := varselect(kernels f, t)
    m:N := reduce(max, [height k for k in l], 0)$List(N)
    for k in l repeat
        height k = m and has?(operator k, ALGOP) => return true
    false

-- differentiate a kernel of the form \( \text{laplace}(l.1,l.2,l.3) \) w.r.t \( x \).
-- note that \( x \) is not necessarily \( l.3 \)
-- if \( x = l.3 \), then there is no use recomputing the laplace transform,
-- it will remain formal anyways

    dvlap(l, x) ==
        l1 := first l
        l2 := second l
        x = (v := retract(l3 := third l)@SE) => - oplap(l2 * l1, l2, l3)
        e := exp(- l3 * l2)
        locallaplace(differentiate(e * l1, x) / e, retract(l2)@SE, l2, v, l3)
isLinear(f, t) ==
ff := univariate(f, kernel(t)@K)
((d := retractIfCan(denom ff)@Union(F, "failed")) case "failed")
or (degree(numer ff) > 1) => "failed"
freeOf?(b := coefficient(numer ff, 0) / d, t) and
freeOf?(c := coefficient(numer ff, 1) / d, t) => [b, c]
"failed"

atn(f, t) ==
if ((v := isExpt f) case Record(var:K, exponent:Integer)) then
w := v::Record(var:K, exponent:Integer)
(w.exponent > 0) and
((vv := symbolIfCan(w.var)) case SE) and (vv::SE = t) =>
return [1, w.exponent::PI]
(u := isTimes f) case List(F) =>
c:F := 1
d:N := 0
for g in u::List(F) repeat
if (rec := atn(g, t)) case Record(coef:F, deg:PI) then
r := rec::Record(coef:F, deg:PI)
c := c * r.coef
d := d + r.deg
else c := c * g
zero? d => "failed"
[c, d::PI]
"failed"

aexp(f, t) ==
if?(f, "exp"::SE) =>
(v := isLinear(first argument(retract(f)@K), t)) case "failed" => "failed"
[1, v.nconst, v.const]
(u := isTimes f) case List(F) =>
c:F := 1
c1 := c0 := 0$F
for g in u::List(F) repeat
if (r := aexp(g, t)) case Record(coef:F, coef1:F, coef0:F) then
rec := r::Record(coef:F, coef1:F, coef0:F)
c := c * rec.coef
c0 := c0 + rec.coef0
c1 := c1 + rec.coef1
else c := c * g
zero? c0 and zero? c1 => "failed"
[c, c1, c0]
if (v := isPower f) case Record(val:F, exponent:Integer) then
w := v::Record(val:F, exponent:Integer)
(w.exponent ^= 1) and
((r := aexp(w.val, t)) case Record(coef:F, coef1:F, coef0:F)) =>
rec := r::Record(coef:F, coef1:F, coef0:F)
return [rec.coef ** w.exponent, w.exponent * rec.coef1,
       w.exponent * rec.coef0]

"failed"

mkPlus f ==
(u := isPlus numer f) case "failed" => "failed"
d := denom f
[p / d for p in u::List(SparseMultivariatePolynomial(R, K))]

-- returns g if f = g/t
tdenom(f, t) ==
(denom f exquo numer t) case "failed" => "failed"
t * f

intlaplace(f, ss, g, v, vv) ==
is?(g, oplap) or ((i := integrate(g, v)) case List(F)) => "failed"
(u:=limit(i::F,equation(vv::OFE,plusInfinity()$OFE)$EQ)) case OFE =>
(1 := limit(i::F, equation(vv::OFE, ss::OFE)$EQ)) case OFE =>
retractIfCan(u::OFE - l::OFE)@Union(F, "failed")
"failed"
"failed"

lapkernel(f, t, tt, ss) ==
(k := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
empty?(arg := argument(k::K)) => "failed"
is?(op := operator k, "%diff"::SE) =>
not( #arg = 3) => "failed"
not(is?(arg.3, t)) => "failed"
fint := eval(arg.1, arg.2, tt)
s := name operator (kernels(ss).1)
ss * locallaplace(fint, t, tt, s, ss) - eval(fint, tt = 0)
not (empty?(rest arg)) => "failed"
member?(t, variables(a := first(arg) / tt)) => "failed"
is?(op := operator k, "Si"::SE) => atan(a / ss) / ss
is?(op, "Ci"::SE) => log((ss**2 + a**2) / a**2) / (2 * ss)
is?(op, "Ei"::SE) => log((ss + a) / a) / ss
-- digamma (or Gamma) needs SpecialFunctionCategory
-- which we do not have here
-- is?(op, "log"::SE) => (digamma(1) - log(a) - log(ss)) / ss
"failed"

-- Below we try to apply one of the textbook rules for computing
-- Laplace transforms, either reducing problem to simpler cases
-- or using one of known base cases
locallaplace(f, t, tt, s, ss) ==
zero? f => 0
one? f => inv ss
(f = 1) => inv ss

laplace(f(t)/t,t,s)
  = integrate(laplace(f(t),t,v), v = s..%plusInfinity)

(x := tdenom(f, tt)) case F =>
g := locallaplace(x::F, t, tt, vv := new()$SE, vvv := vv::F)
(x := intlaplace(f, ss, g, vv, vvv)) case F => x::F
oplap(f, tt, ss)

Use linearity
(u := mkPlus f) case List(F) =>
  +/[locallaplace(g, t, tt, s, ss) for g in u::List(F)]
(rec := splitConstant(f, t)).const ^= 1 =>
  rec.const * locallaplace(rec.nconst, t, tt, ss)

laplace(t^n*f(t),t,s) = (-1)^n*D(laplace(f(t),t,s), s, n)

(v := atn(f, t)) case Record(coef:F, deg:PI) =>
  vv := v::Record(coef:F, deg:PI)
  is?(la := locallaplace(vv.coef, t, tt, s, ss), oplap) => oplap(f,tt,ss)
  (-1$Integer)**(vv.deg) * differentiate(la, s, vv.deg)

Complex shift rule
(w := aexp(f, t)) case Record(coef:F, coef1:F, coef0:F) =>
  ww := w::Record(coef:F, coef1:F, coef0:F)
  exp(ww.coef0) * locallaplace(ww.coef,t,tt,s,ss - ww.coef1)

Try base cases
(x := lapkernel(f, t, tt, ss)) case F => x::F

The following does not seem to help computing transforms, but
-- quite frequently leads to loops, so I (wh) disabled it for now
-- last chance option: try to use the fact that
-- laplace(f(t),t,s) = s laplace(g(t),t,s) - g(0) where dg/dt = f(t)
-- elem?(int := lfintegrate(f, t)) and (rint := retractIfCan int) case F =>
-- fint := rint :: F
-- to avoid infinite loops, we don’t call laplace recursively
-- if the integral has no new logs and f is an algebraic function
-- empty?(logpart int) and algebraic?(f, t) => oplap(fint, tt, ss)
-- ss * locallaplace(fint, t, tt, s, ss) - eval(fint, tt = 0)
-- oplap(f, tt, ss)

setProperty(oplap,SPECIALDIFF,dvlap@((List F,SE)->F) pretend None)
package LAZM3PK LazardSetSolvingPackage

— LazardSetSolvingPackage.input —

)set break resume
)sys rm -f LazardSetSolvingPackage.output
)spool LazardSetSolvingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 36
R := Integer
--R
--R (1) Integer
--R Type: Domain
--E 1

--S 2 of 36
ls : List Symbol := [b1,x,y,z,t,v,u,w]
--R
--R (2) [b1,x,y,z,t,v,u,w]
--R Type: List(Symbol)
--E 2

--S 3 of 36
V := OVAR(ls)
--R
--R (3) OrderedVariableList([b1,x,y,z,t,v,u,w])
--R Type: Domain
--E 3

--S 4 of 36
E := IndexedExponents V
--R
--R (4) IndexedExponents(OrderedVariableList([b1,x,y,z,t,v,u,w]))
```
```
```
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 10

--S 11 of 36
u: P := 'u
--R
--R (11) u
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 11

--S 12 of 36
v: P := 'v
--R
--R (12) v
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 12

--S 13 of 36
w: P := 'w
--R
--R (13) w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 13

--S 14 of 36
T := REGSET(R,E,V,P)
--R
--R (14)
--R RegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([b1,x,y,z,t
--R ,v,u,w]),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePolyno
--R mial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])))
--R Type: Domain
--E 14

--S 15 of 36
p0 := b1 + y + z - t - w
--R
--R (15) b1 + y + z - t - w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 15

--S 16 of 36
p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1
--R
--R
|--R 2
|--R (16) 2v y + 2u z + 2w t - 2w - w - 1
|--E 16

|--S 17 of 36
p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w
|--R
|--R
|--R (17) 3v y + 3u z + (-3w - 1)t + 3w + 3w + 4w
|--E 17

|--S 18 of 36
p3 := 6*x*z*v - 6*t*w**2 + 6*w**3 - 3*t*w + 6*w**2 - t + 4*w
|--R
|--R
|--R (18) 6v z x + (-6w - 3w - 1)t + 6w + 6w + 4w
|--E 18

|--S 19 of 36
p4 := 4*z*u**3 + 4*y*v**3 + 4*t*w**3 - 4*w**4 - 6*w**3 + 4*t*w - 10*w**2 - w - 1
|--R
|--R
|--R (19) 4v y + 4u z + (4w + 4w)t - 4w - 6w - 10w - w - 1
|--E 19

|--S 20 of 36
p5 := 8*x*z*u*v + 8*t*w**3 - 8*w**4 + 4*t*w**2 - 12*w**3 - 4*t*w - 10*w**2 - 3*w - 1
|--R
|--R
|--R (20) 8v z x + (8w + 4w + 4w)t - 8w - 12w - 14w - 3w - 1
|--E 20

|--S 21 of 36
p6 := 12*x*z*v**2 + 12*t*w**3 - 12*w**4 + 12*t*w**2 - 18*w**3 + 8*t*w - 14*w**2 - w - 1
|--R
|--R
|--R (21) 12v z x + (12w + 12w + 8w)t - 12w - 18w - 14w - w - 1
|--E 21
--S 22 of 36
p7 := -24*t*w**3 + 24*w**4 - 24*t*w**2 + 36*w**3 - 8*t*w + 26*w**2 + 7*w + 1
--R
--R
--R (22) (- 24w - 24w - 8w)t + 24w + 36w + 26w + 7w + 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 22

--S 23 of 36
lp := [p0, p1, p2, p3, p4, p5, p6, p7]
--R
--R
--R (23)
--R 2
--R [b1 + y + z - t - w, 2v y + 2u z + 2w t - 2w - w - 1,
--R 2 2 2 3 2
--R 3v y + 3u z + (- 3w - 1)t + 3w + 3w + 4w,
--R 2 3 3 2
--R 6v z x + (- 6w - 3w - 1)t + 6w + 6w + 4w,
--R 3 3 3 4 3 2
--R 4v y + 4u z + (4w + 4w)t - 6w - 6w - 10w - w - 1,
--R 3 2 4 3 2
--R 8u v z x + (8w + 4w + 4w)t - 8w - 12w - 14w - 3w - 1,
--R 2 3 2 4 3 2
--R 12v z x + (12w + 12w + 8w)t - 12w - 18w - 14w - w - 1,
--R 3 2 4 3 2
--R (- 24w - 24w - 8w)t + 24w + 36w + 26w + 7w + 1]
--IType: List(NewSparseMultivariatePolynomial(Integer,...
--E 23

--S 24 of 36
lts := zeroSetSplit(lp,false)$T
--R
--R
--R (24)
--R {[w + 1,u,v,t + 1,b1 + y + z + 2], {w + 1,v,t + 1,z,b1 + y + 2},
--R [w + 1,t + 1,z,y,b1 + 2], {w + 1,v - u,t + 1,y + z,x,b1 + 2},
--R [w + 1,u,t + 1,y,x,b1 + z + 2],
--R 5 4 3 2
--R {144w + 216w + 96w + 6w - 11w - 1,
--R 2 2 5 4 3 2
--R (12w + 9w + 1)u - 72w - 108w - 42w - 9w - 3w,
--R 2 4 3 2
--R (12w + 9w + 1)v + 36w + 54w + 18w,
--R 3 2 4 3 2
--R (24w + 24w + 8w)t - 24w - 36w - 26w - 7w - 1,
--R 2 2 4 3 2

--E 24
CHAPTER 13. CHAPTER L

\[ (12u v - 12u)z + (12w v + 12w + 4)t + (3w - 5)v + 36w + 42w + 6w \]
\[ + 16w \]
\[ 2v y + 2u z + 2w t - 2w - w - 1, \]
\[ 6v z x + (-6w - 3w - 1)t + 6w + 6w + 4w, b1 + y + z - t - w \]

IType: List(RemarkableTriangularSet(Integer,...

S 25 of 36
[coHeight(ts) for ts in lts]
--R
--S 26 of 36
ST := SREGSET(R,E,V,P)
--R
--S 27 of 36
pack := LAZM3PK(R,E,V,P,T,ST)
--R
--S 28 of 36
zeroSetSplit(lp,false)$pack
---R
---R (28)
---R {{w + 1,t + 1,z,y,b1 + 2}, {w + 1,v,t + 1,z,b1 + y + 2},
---R {w + 1,u,v,t + 1,z,b1 + y + z + 2}, {w + 1,v - u,t + 1,y + z,x,b1 + 2},
---R {w + 1,u,t + 1,y,x,b1 + z + 2},
---R
---R 5 4 3 2 4 3 2
---R {144w + 216w + 96w + 6w - 11w - 1, u - 24w - 36w - 14w + w + 1,
---R 4 3 2 4 3 2
---R 3v - 48w - 60w - 10w + 8w + 2, t - 24w - 36w - 14w - w + 1,
---R 4 3 2
---R 486z - 2772w - 4662w - 2055w + 30w + 127,
---R 4 3 2
---R 2916y - 22752w - 30312w - 8220w + 2064w + 1561,
---R 4 3 2
---R 356x - 3696w - 4536w - 968w + 822w + 371,
---R 4 3 2
---R 2916b1 - 30600w - 46692w + 20274w + 593}
---R ]
---IType: List(SquareFreeRegularTriangularSet(Integer,...
---E 28

---S 29 of 36
f0 := (w - v) ** 2 + (u - t) ** 2 - 1
---R
---R 2 2 2 2
---R (29) t - 2u t + v - 2w v + u + w - 1
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 29

---S 30 of 36
f1 := t ** 2 - v ** 3
---R
---R 2 3
---R (30) t - v
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 30

---S 31 of 36
f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)
---R
---R 2 2
---R (31) (- 3v - 2v + 2w)t + 3u v
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 31
\[ f_3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1) \]
\[ \begin{align*}
&2 &2 &2 \\
&6v t z + (-2t - 3v)z + 1
\end{align*} \]
\[ \text{Type: NewSparseMultivariatePolynomial(Integer,...) --E 32} \]

\[ l_f := [f_0, f_1, f_2, f_3] \]
\[ \begin{align*}
&2 &2 &2 &2 &3 &2 &2 \\
&t - 2u t + v - 2w v + u + w - 1, t - v, (-3v - 2v + 2w)t + 3u v,
\end{align*} \]
\[ \begin{align*}
&2 &2 &2 \\
&6v t z + (-2t - 3v)z + 1
\end{align*} \]
\[ \text{Type: List(NewSparseMultivariatePolynomial(Integer,...) --E 33} \]

\[ \text{zeroSetSplit}(l_f, \text{true})\]$T$
\[ \begin{align*}
&6 &3 &2 &4 \\
&729u + (-1458w + 729w - 4158w - 1685)u
\end{align*} \]
\[ \begin{align*}
&6 &5 &4 &3 &2 &2 &8 \\
&(729w - 1458w - 2619w + 4892w - 297w + 5814w + 427)u + 729w
\end{align*} \]
\[ \begin{align*}
&7 &6 &5 &4 &3 &2 \\
&216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529
\end{align*} \]
\[ \begin{align*}
&4 &3 &2 &2 &6 &5 \\
&2187u + (-4374w - 972w - 12474w - 2868)u + 2187w - 1944w
\end{align*} \]
\[ \begin{align*}
&4 &3 &2 \\
&-10125w - 4800w + 2501w + 4968w - 1587
\end{align*} \]
\[ \begin{align*}
&3 &2 &2 &6 &5 &4 &3 &2 \\
&(1944w - 108w)u + 972w + 3024w - 1080w + 496w + 1116w
\end{align*} \]
\[ \begin{align*}
&2 &2 &2 &2 &2
\end{align*} \]
-R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
-\:] 
-\] 
--IType: List(RegularTriangularSet(Integer, IndexedExponents(...
--E 34

--S 35 of 36
zeroSetSplit(If, false)$T
--R
--R
--R (35)
--R [
--R { 
--R 6 3 2 4
--R 729u + (- 1458w + 729w - 4158w - 1685)v
--R +
--R 6 5 4 3 2 2 8
--R (729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u + 729w
--R +
--R 7 6 5 4 3 2
--R 216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529
--R ,
--R
--R 4 3 2 2 6 5
--R 2187u + (- 4374w - 972w - 12474w - 2868)u + 2187w - 1944w
--R +
--R 4 3 2
--R - 10125w - 4800w + 2501w + 4968w - 1587
--R *
--R *
--R +
--R 3 2 2 6 5 4 3 2
--R (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w
--R ,
--R 2 2
--R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
--R,
--R
--R 4 3 2
--R {27w + 4w - 54w - 36w + 23, u, (12w + 2)v - 9w - 2w + 9,
--R 2 2
--R 6t - 2v - 3w + 2w + 3, 2t z - 1}
--R ,
--R
--R 6 5 4 3 2
--R {59049w + 91854w - 45198w + 145152w + 63549w + 60922w + 21420,
--R ,
--R 5 4 3
--R 31484448266904w - 18316865522574w + 23676995746098w
--R +
--R 2
\[\begin{align*}
\text{Type: } & \text{List(RegularTriangularSet(Integer,IndexedExponents(...))} \\
\text{S 36 of 36} \\
\text{zeroSetSplit(lf,false)} \text{pack} \\
\text{(36)} \\
\text{[]} \\
\text{[} \\
\text{6} & 3 & 2 & 4 \\
\text{729u} & + & (-1458w + 729w - 4158w - 1685)u \\
\text{+} \\
\text{6} & 5 & 4 & 3 & 2 & 2 & 8 \\
\text{(729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u} & + & 729w \\
\text{+} \\
\text{7} & 6 & 4 & 3 & 2 \\
\text{216w} & - & 2900w & - & 2376w & + & 3870w & + & 4072w & - & 1188w & - & 1656w & + & 529 \\
\text{,} \\
\text{[} \\
\text{4} & 3 & 2 & 2 & 6 & 5 \\
\text{2187u} & + & (-4374w - 972w - 12474w - 2868)u} & + & 2187w & - & 1944w \\
\text{+} \\
\text{4} & 3 & 2 \\
\text{-10125w} & - & 4800w & + & 2501w & + & 4968w & - & 1587 \\
\text{*,} \\
\text{v} \\
\text{+} \\
\end{align*}\]
--R 3 2 2 6 5 4 3 2
--R (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w
--R ,
--R 2 2 2 2 2
--R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
--R ,
--R 2 2 2
--R {81w + 18w + 28, 729u - 1890w - 533, 81v + (- 162w + 27)v - 72w - 112,
--R 1188t + (972w + 2997)u v + (- 11448w - 11536)u,
--R 2
--R 641237934604288z
--R +
--R (78614584763904w + 26785578742272)u + 23614361865616w
--R +
--R 70221988585728
--R *
--R v
--R +
--R (358520253138432w + 101922133759488)u + 142598803536000w
--R +
--R 54166419595008
--R *
--R z
--R +
--R (32655103844499w - 44224572465882)u v
--R +
--R (43213900115457w - 32432039102070)u
--R }
--R ,
--R 4 3 2 3 2
--R {27w + 4w - 54w - 36w + 23, u, 218v - 162w + 3w + 160w + 153,
--R 2 3 2 3 2
--R 109t - 27w - 54w + 63w + 80, 1744z + (- 1458w + 27w + 1440w + 505)t}
--R ,
--R 4 3 2
--R {27w + 4w - 54w - 36w + 23, u, 218v - 162w + 3w + 160w + 153,
--R 2 3 2
--R 109t - 27w - 54w + 63w + 80, 1308z + 162w - 3w - 814w - 153}
--R ,
--R 4 3 2 2 2
--R {729w + 972w - 1026w + 1684w + 765, 81u + 72w + 16w - 72,
--R 3 2
--R 702v - 162w - 225w + 40w - 99,
--R 3 2
--R 11336t + (324w - 603w - 1718w - 1557)u,
A package for solving polynomial systems by means of Lazard triangular sets. This package provides two operations. One for solving in the sense of the regular zeros, and the other for solving in the sense of the Zariski closure. Both produce square-free regular sets. Moreover, the decompositions do not contain any redundant component. However, only zero-dimensional regular sets are normalized, since normalization may be time consuming in positive dimension.

The decomposition process is that of


The LazardSetSolvingPackage package constructor solves polynomial systems by means of Lazard triangular sets. However one condition is relaxed: Regular triangular sets whose saturated ideals have positive dimension are not necessarily normalized.

The decompositions are computed in two steps. First the algorithm of Moreno Maza (implemented in the RegularTriangularSet domain constructor) is called. Then the resulting decompositions are
converted into lists of square-free regular triangular sets and the redundant components are removed. Moreover, zero-dimensional regular triangular sets are normalized.

Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

The LazardSetSolvingPackage constructor takes six arguments. The first one, \( R \), is the coefficient ring of the polynomials; it must belong to the category \( \text{GcdDomain} \). The second one, \( E \), is the exponent monoid of the polynomials; it must belong to the category \( \text{OrderedAbelianMonoidSup} \). The third one, \( V \), is the ordered set of variables; it must belong to the category \( \text{OrderedSet} \). The fourth one is the polynomial ring; it must belong to the category \( \text{RecursivePolynomialCategory}(R,E,V) \). The fifth one is a domain of the category \( \text{RegularTriangularSetCategory}(R,E,V,P) \) and the last one is a domain of the category \( \text{SquareFreeRegularTriangularSetCategory}(R,E,V,P) \). The abbreviation for LazardSetSolvingPackage is \( \text{LAZM3PK} \).

For the purpose of solving zero-dimensional algebraic systems, see also \( \text{LexTriangularPackage} \) and \( \text{ZeroDimensionalSolvePackage} \). These packages are easier to call than \( \text{LAZM3PK} \). Moreover, the \( \text{ZeroDimensionalSolvePackage} \) package provides operations to compute either the complex roots or the real roots.

We illustrate now the use of the LazardSetSolvingPackage package constructor with two examples (Butcher and Vermeer).

Define the coefficient ring.

\[
R := \text{Integer}
\]

Define the list of variables,

\[
ls : \text{List Symbol} := [b1,x,y,z,t,v,u,w]
\]

and make it an ordered set:

\[
V := \text{OVAR}(ls)
\]

then define the exponent monoid.

\[
E := \text{IndexedExponents}(V)
\]

Define the polynomial ring.

\[
P := \text{NSMP}(R, V)
\]
Let the variables be polynomial.

\begin{verbatim}
b1: P := 'b1
x: P := 'x
y: P := 'y
z: P := 'z
t: P := 't
u: P := 'u
v: P := 'v
w: P := 'w
\end{verbatim}

Now call the \tt{RegularTriangularSet} domain constructor.

\begin{verbatim}
T := REGSET(R,E,V,P)
\end{verbatim}

Define a polynomial system (the Butcher example).

\begin{verbatim}
p0 := b1 + y + z - t - w
p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1
p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w
p3 := 6*x*z*v - 6*t*u**2 + 6*u**3 - 3*t*w + 6*u**2 - t + 4*w
p4 := 4*z*u**3 + 4*y*v**3 + 4*t*w**3 - 4*u**4 - 6*u**3 + 4*t*u - 10*u**2 - w - 1
p5 := 8*x*z*v**2 + 8*y*v**2 - 8*t*u**2 - 4*t*w**3 - 4*t**2 - 14*u**2 - 3*t**2 - 1
p6 := 12*x*z*v**2 + 12*t*w**3 - 12*u**4 + 12*t**2 + 16*u**3 + 8*t**2 - 14*w**2 - w - 1
p7 := -24*t**2 - 24*u**4 - 24*t**2 + 36*u**3 + 8*t**2 + 26*u**2 + 7*w + 1
lp := [p0, p1, p2, p3, p4, p5, p6, p7]
\end{verbatim}
First of all, let us solve this system in the sense of Lazard by means of the REGSET constructor:

\[
lts := \text{zeroSetSplit}(lp, false) \{\{w + 1, u, v, t + 1, b1 + y + z + 2\}, \{w + 1, v, t + 1, z, b1 + y + 2\},
\{w + 1, t + 1, z, y, b1 + 2\}, \{w + 1, u, t + 1, y, z, b1 + 2\},
\{w + 1, u, t + 1, y, x, b1 + z + 2\},
\{144u + 216u + 96u + 6u - 11u - 1, 12u + 9u + 1\}u - 72u - 108u - 42u - 9u - 3u,
\{12u + 9u + 1\}v + 36u + 54u + 18u , 24u + 24u + 8u)t - 24u - 36u - 26u - 7u - 1, 2(12v - 12v )z + (12w v + 12w + 4)t + (3w - 5)v + 36w + 42w + 6w + 16w,
\{2v + 2u z + 2w t - 2w - w - 1, 6v x + (- 6w - 3w - 1)t + 6w + 6w + 4w, b1 + y + z - t - w\}\}
\]

We can get the dimensions of each component of a decomposition as follows.

[coHeight(ts) for ts in lts]
[3,3,3,2,0]

The first five sets have a simple shape. However, the last one, which has dimension zero, can be simplified by using Lazard triangular sets.

Thus we call the SquareFreeRegularTriangularSet domain constructor,

\[
\text{ST := SREGSET}(R,E,V,P)
\]

and set the LAZM3PK package constructor to our situation.

\[
\text{pack := LAZM3PK}(R,E,V,P,T,ST)
\]
We are ready to solve the system by means of Lazard triangular sets:

\[
\text{zeroSetSplit}(lp,\text{false})\text{pack}
\]

\[
\{w + 1, t + 1, z, y, b1 + 2\}, \{w + 1, v, t + 1, z, b1 + y + 2\}, \{w + 1, u, v, t + 1, y + z, x, b1 + 2\},
\]

\[
\{w + 1, u, t + 1, y, x, b1 + z + 2\},
\]

\[
5 \quad 4 \quad 3 \quad 2 \quad 4 \quad 3 \quad 2
\]

\[
\{144w + 216w + 96w + 6w - 11w - 1, u - 24w - 36w - 14w + w + 1,
\]

\[
4 \quad 3 \quad 2 \quad 4 \quad 3 \quad 2
\]

\[
3v - 48w - 60w - 10w + 8w + 2, t - 24w - 36w - 14w - w + 1,
\]

\[
4 \quad 3 \quad 2
\]

\[
486z - 2772w - 4662w - 2055w + 30w + 127,
\]

\[
4 \quad 3 \quad 2
\]

\[
2916y - 22752w - 30312w - 8220w + 2064w + 1561,
\]

\[
4 \quad 3 \quad 2
\]

\[
356x - 3696w - 4536w - 968w + 822w + 371,
\]

\[
4 \quad 3 \quad 2
\]

\[
2916b1 - 30600w - 46692w - 20274w - 8076w + 593\}
\]

We see the sixth triangular set is nicer now: each one of its polynomials has a constant initial.

We follow with the Vermeer example. The ordering is the usual one for this system.

Define the polynomial system.

\[
f0 := (w - v) ** 2 + (u - t) ** 2 - 1
\]

\[
t - 2u t + v - 2w v + u + w - 1
\]

\[
f1 := t ** 2 - v ** 3
\]

\[
t - v
\]

\[
f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)
\]

\[
(- 3v - 2v + 2w)t + 3u v
\]

\[
f3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1)
\]

\[
2 \quad 2
\]
First of all, let us solve this system in the sense of Kalkbrener by means of the REGSET constructor:

```latex
zeroSetSplit(lf, true)$T
```

We have obtained one regular chain (i.e. regular triangular set) with dimension 1. This set is in fact a characterist set of the (radical of) the ideal generated by the input system \( \text{lf} \). Thus we have only the generic points of the variety associated with \( \text{lf} \) (for the elimination ordering given by \( \text{ls} \)).

So let us get now a full description of this variety.

Hence, we solve this system in the sense of Lazard by means of the REGSET constructor:
zeroSetSplit(1, false)$T$

\[
\begin{align*}
&\{ \\
&\quad \begin{array}{cccccc}
6 & 3 & 2 & 4 \\
729u & + & (-1458w + 729w - 4158w - 1685)u \\
+ & \\
6 & 5 & 4 & 3 & 2 & 2 & 8 \\
(729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u & + & 729w \\
+ & \\
7 & 6 & 5 & 4 & 3 & 2 \\
216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529 \\
\right. \\
, \\
4 & 3 & 2 & 2 & 6 & 5 \\
2187u & + & (-4374w - 972w - 12474w - 2868)u & + & 2187w - 1944w \\
+ & \\
4 & 3 & 2 \\
- 10125w - 4800w & + & 2501w & + & 4968w - 1587 \\
* v \\
+ & \\
3 & 2 & 2 & 6 & 5 & 4 & 3 & 2 \\
(1944w - 108w )u & + & 972w & + & 3024w & - & 1080w & + & 496w & + & 1116w \\
, \\
2 & 2 & 2 & 2 & 2 \\
(3v + 2v - 2w)t & - & 3u v & , \ ((4v - 4w)t - 6u v )z & + & (2t + 3v )z - 1 \\
, \\
4 & 3 & 2 \\
\{27w + 4w - 54w - 36w + 23, u, (12w + 2)v - 9w - 2w + 9, \\
2 & 2 \\
6t - 2v - 3w & + & 2w + 3, 2t z - 1 \} \\
, \\
6 & 5 & 4 & 3 & 2 \\
\{59049w + 91854w - 45198w + 145152w + 63549w + 60922w + 21420, \\
5 & 4 & 3 \\
31484448266904w & - & 18316865522574w & + & 23676995746098w \\
+ & \\
2 & 6657857188965w & + & 8904703998546w & + & 3890631403260 \\
* \\
2 & \\
u & + \\
5 & 4 & 3 \\
94262810316408w & - & 82887296576616w & + & 89801831438784w \\
+ \\
\}
\end{align*}
\]
We retrieve our regular chain of dimension 1 and we get three regular chains of dimension 0 corresponding to the degenerated cases. We want now to simplify these zero-dimensional regular chains by using Lazard triangular sets. Moreover, this will allow us to prove that the above decomposition has no redundant component.

Generally, decompositions computed by the REGSET constructor do not have redundant components. However, to be sure that no redundant component occurs one needs to use the SREGSET or LAZM3PK constructors.

So let us solve the input system in the sense of Lazard by means of the LAZM3PK constructor:

```plaintext
zeroSetSplit(lf,false)$pack
[
  
  6 3 2 4
  729u + (- 1458w + 729w - 4158w - 1685)u
  +
  6 5 4 3 2 2 8
  (729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u + 729w
  +
  7 6 5 4 3 2
  216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529

  4 3 2 2
  2187u + (- 4374w - 972w - 12474w - 2868)u + 2187w - 1944w
  +
  4 3 2
  - 10125w - 4800w + 2501w + 4968w - 1587
* v
+ 
```
\[
\begin{align*}
& (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w, \\
& (3v + 2v - 2w)t - 3u v, ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1, \\
& (81w + 18w + 28, 729u - 1890w - 533, 81v + (- 162w + 27)v - 72w - 112, \\
& 11881t + (972w + 2997)u v + (- 11448w - 11536)u, \\
& 641237934604288z \\
& + (78614584763904w + 26785578742272)u + 236143618655616w \\
& + 70221988585728 \\
& * v \\
& + (35852025318432w + 10192213759488)u + 142598803536000w \\
& + 54166419595008 \\
& * z \\
& + (32655103844499w - 44224572465882)u v \\
& + (43213900115457w - 32432039102070)u.
\end{align*}
\]
Due to square-free factorization, we obtained now four zero-dimensional regular chains. Moreover, each of them is normalized (the initials are constant). Note that these zero-dimensional components may be investigated further with the ZeroDimensionalSolvePackage package constructor.

See also:
- )show LazardSetSolvingPackage
- )show ZeroDimensionalSolvePackage

LazardSetSolvingPackage (LAZM3PK)

Exports:
- normalizeIfCan
- zeroSetSplit
package LAZM3PK LazardSetSolvingPackage

)abbrev package LAZM3PK LazardSetSolvingPackage
++ Author: Marc Moreno Maza
++ Date Created: 10/02/1998
++ Date Last Updated: 12/16/1998
++ References :
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ Description:
++ A package for solving polynomial systems by means of Lazard triangular
++ sets. This package provides two operations. One for solving in the sense
++ of the regular zeros, and the other for solving in the sense of
++ the Zariski closure. Both produce square-free regular sets.
++ Moreover, the decompositions do not contain any redundant component.
++ However, only zero-dimensional regular sets are normalized, since
++ normalization may be time consumming in positive dimension.
++ The decomposition process is that of [2].

LazardSetSolvingPackage(R,E,V,P,TS,ST): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS: RegularTriangularSetCategory(R,E,V,P)
ST : SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Split ==> List TS
--KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
--EntryGcd ==> List PWT
--HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
--KeyInvSet ==> Record(arg1: P, arg3: TS)
--EntryInvSet ==> List TS
--HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,ST)
normalizpack ==> NormalizationPackage(R,E,V,P,ST)
Exports == with

normalizeIfCan: ST -> ST
  ++ \texttt{normalizeIfCan(ts)} returns \texttt{ts} in an normalized shape
  ++ if \texttt{ts} is zero-dimensional.
zeroSetSplit: (LP, B) -> List ST
  ++ \texttt{zeroSetSplit(lp,clos?)} has the same specifications as
  ++ \texttt{zeroSetSplit(lp,clos?) from RegularTriangularSetCategory}.

Implementation == add

convert(st: ST): TS ==
  ts: TS := empty()
  lp: LP := members(st)$ST
  lp := sort(infRittWu?,lp)
  for p in lp repeat
    ts := internalAugment(p,ts)$TS
  ts

squareFree(ts: TS): List ST ==
  empty? ts => [empty()$ST]
  lp: LP := members(ts)$TS
  lp := sort(infRittWu?,lp)
  newts: ST := empty()$ST
  toSee: List ST := [newts]
  toSave: List ST
  for p in lp repeat
    toSave := []
    while (not empty? toSee) repeat
      us := first toSee; toSee := rest toSee
      lpwt := stoseSquareFreePart(p,us)$regsetgcdpack
      for pwt in lpwt repeat
        newus := internalAugment(pwt.val,pwt.tower)$ST
        toSave := cons(newus,toSave)
      toSee := toSave
    toSave
  toSave

normalizeIfCan(ts: ST): ST ==
  empty? ts => ts
  lp: LP := members(ts)$ST
  lp := sort(infRittWu?,lp)
  p: P := first lp
  not univariate?(p)$polsetpack => ts
  lp := rest lp
  newts: ST := empty()$ST
  newts := internalAugment(p,newts)$ST
  while (not empty? lp) repeat
    p := first lp
    lv := variables(p)
for v in lv repeat
  v = mvar(p) => "leave"
  not algebraic?(v,newts) => return internalAugment(lp,newts)$ST
lp := rest lp
p := normalizedAssociate(p,newts)$normalizpack
newts := internalAugment(p,newts)$ST
newts

zeroSetSplit(lp:List(P), clos?:B): List ST ==
  -- if clos? then SOLVE in the closure sense
toSee: Split := zeroSetSplit(lp, clos?)$TS
toSave: List ST := []
for ts in toSee repeat
  toSave := concat(squareFree(ts),toSave)
toSave := removeSuperfluousQuasiComponents(toSave)$quasicomppack
  [normalizeIfCan(ts) for ts in toSave]

package LEADCDET LeadingCoefDetermination

  — LeadingCoefDetermination.input —

)set break resume
)sys rm -f LeadingCoefDetermination.output
)spool LeadingCoefDetermination.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LeadingCoefDetermination
--E 1

)spool
)lisp (bye)
---

--- LeadingCoefDetermination.help ---

====================================================================
LeadingCoefDetermination examples
====================================================================

Package for leading coefficient determination in the lifting step.
Package working for every R euclidean with property "F".

See Also:

o )show LeadingCoefDetermination

---

LeadingCoefDetermination (LEADCDET)

---

Exports:

   distFact    polCase

--- package LEADCDET LeadingCoefDetermination ---

)abbrev package LEADCDET LeadingCoefDetermination
++ Author : P.Gianni, May 1990
++ Description:
++ Package for leading coefficient determination in the lifting step.
++ Package working for every R euclidean with property "F".

LeadingCoefDetermination(OV,E,Z,P) : C == T
where
  OV    : OrderedSet
  E     : OrderedAbelianMonoidSup
Z : EuclideanDomain
BP ==> SparseUnivariatePolynomial Z
P : PolynomialCategory(Z,E,OV)
NNI ==> NonNegativeInteger
LeadFact ==> Record(polfac:List(P),correct:Z,corrfact:List(BP))
ParFact ==> Record(irr:P,pow:Integer)
FinalFact ==> Record(contp:Z,factors:List(ParFact))

C == with
polCase : (Z,NNI,List(Z)) -> Boolean
++ polCase(contprod, numFacts, evallcs), where contprod is the
++ product of the content of the leading coefficient of
++ the polynomial to be factored with the content of the
++ evaluated polynomial, numFacts is the number of factors
++ of the leadingCoefficient, and evallcs is the list of
++ the evaluated factors of the leadingCoefficient, returns
++ true if the factors of the leading Coefficient can be
++ distributed with this valuation.
distFact : (Z,List(BP),FinalFact,List(Z),List(OV),List(Z)) ->
  Union(LeadFact,"failed")
++ distFact(contm,unilist,plead,vl,lvar,lval), where contm is
++ the content of the evaluated polynomial, unilist is the list
++ of factors of the evaluated polynomial, plead is the complete
++ factorization of the leading coefficient, vl is the list
++ of factors of the leading coefficient evaluated, lvar is the
++ list of variables, lval is the list of values, returns a record
++ giving the list of leading coefficients to impose on the univariate
++ factors,

T == add
distribute: (Z,List(BP),List(P),List(Z),List(OV),List(Z)) -> LeadFact
checkpow : (Z,Z) -> NNI

polCase(d:Z,nk:NNI,lval:List(Z)):Boolean ==
  -- d is the product of the content lc m (case polynomial)
  -- and the cont of the polynomial evaluated
  q:Z
  distlist:List(Z) := [d]
  for i in 1..nk repeat
    q := unitNormal(lval.i).canonical
    for j in 0..(i-1):NNI repeat
      y := distlist.(i-j)::NNI
      while y^=1 repeat
        y := gcd(y,q)
        q := q quo y
      if q=1 then return false
      distlist := append(distlist,[q])
  true

checkpow(a:Z,b:Z) : NonNegativeInteger ==
qt: Union(Z,"failed")
for i in 0.. repeat
  qt := b exquo a
  if qt case "failed" then return i
b := qt::Z

distribute(contm:Z,unilist:List(BP),pl:List(P),vl:List(Z),
  lvar:List(OV),lval:List(Z)): LeadFact ==
  d, lcp : Z
  nf: NNI:=#unilist
  for i in 1..nf repeat
    lcp := leadingCoefficient (unilist.i)
    d := gcd(lcp,vl.i)
    pl.i := (lcp quo d)*pl.i
    d := vl.i quo d
    unilist.i := d*unilist.i
    contm := contm quo d
    if contm ^=1 then for i in 1..nf repeat pl.i := contm*pl.i
    [pl,contm,unilist]$LeadFact

distFact(contm:Z,unilist:List(BP),plead:FinalFact,
  vl:List(Z),lvar:List(OV),lval:List(Z)):Union(LeadFact,"failed") ==
  h: NonNegativeInteger
  c,d : Z
  lpol:List(P):=[ ]
  lexp:List(Integer):=[ ]
  nf: NNI := #unilist
  vl := reverse vl -- lpol and vl reversed so test from right to left
  for fpl in plead.factors repeat
    lpol:= [fpl.irr,:lpol]
    lexp:= [fpl.pow,:lexp]
    vlp:List(Z):= [1$Z for i in 1..nf]
    aux : List(P) := [1$P for i in 1..nf]
    for i in 1..nf repeat
      c := contm*leadingCoefficient unilist.i
      c=1 or c=-1 => "next i"
      for k in 1..(# lpol) repeat
        lexp.k=0 => "next factor"
        h:= checkpow(vl.k,c)
        if h=0 then
          if h>lexp.k then return "failed"
        lexp.k:=lexp.k-h
        aux.i := aux.i*(lpol.k ** h)
        d:= vl.k**h
        vlp.i:= vlp.i*d
        c:= c quo d
        if contm=1 then vlp.i:=c
      for k in 1..(# lpol) repeat if lexp.k ^= 0 then return "failed"
      contm =1 => [[vlp.i*aux.i for i in 1..nf],1,unilist]$LeadFact
      distribute(contm,unilist,aux,vlp,lvar,lval)
package LEXTRIPK LexTriangularPackage

)set break resume
)spool LexTriangularPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 22
R := Integer
--R
--R
--R (1) Integer
--R
--E 1

--S 2 of 22
ls : List Symbol := [a,b,c,d,e,f]
--R
--R
--R (2) [a,b,c,d,e,f]
--R
--E 2

--S 3 of 22
V := OVAR(ls)
--R
--R
--R
--R (3) OrderedVariableList([a,b,c,d,e,f])
--R
--E 3
\[
P := \text{NSMP}(R, V)
\]

\[
\text{NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))}
\]

\[
p_1 := a*b*c*d*e*f - 1
\]

\[
p_2 := a*b*c*d*e + a*b*c*e*f + a*b*d*e*f + a*c*d*e*f + b*c*d*e*f
\]

\[
p_3 := a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f
\]

\[
p_4 := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f
\]

\[
p_5 := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
\text{NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))}
\]

\[
p_6 := a*b*c*d*e*f - 1
\]

\[
p_7 := a*b*c + a*b*d + a*c*d + b*c*d + c*d*e + d*e*f
\]

\[
p_8 := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_9 := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{10} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
\text{NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))}
\]

\[
p_{11} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{12} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{13} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{14} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{15} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{16} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{17} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{18} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{19} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{20} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{21} := a*b + a*f + b*c + c*d + d*e + e*f
\]

\[
p_{22} := a*b + a*f + b*c + c*d + d*e + e*f
\]
\[ P := a + b + c + d + e + f \]

\[ \text{-R} \]

\[ \text{--R} \]

\[ \text{--R (10) a + b + c + d + e + f} \]

\[ \text{--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))} \]

\[ \text{--E 10} \]

\[ \text{--S 11 of 22} \]

\[ \text{lp := [p1, p2, p3, p4, p5, p6]} \]

\[ \text{--R} \]

\[ \text{--R (11)} \]

\[ \text{[f e d c b a - 1, (((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b,} \]

\[ \text{((d + f)c + f e)b + f e d)a + e d c b + f e d c,} \]

\[ ((c + f)b + f e)a + d c b + e d c + f e d,} \]

\[ (b + f)a + c b + d c + e d + f e, a + b + c + d + e + f]} \]

\[ \text{--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))} \]

\[ \text{--E 11} \]

\[ \text{--S 12 of 22} \]

\[ \text{lextripack := LEXTRIPK(R,ls)} \]

\[ \text{--R} \]

\[ \text{--R (12) LexTriangularPackage(Integer, [a,b,c,d,e,f])} \]

\[ \text{--R Type: Domain} \]

\[ \text{--E 12} \]

\[ \text{--S 13 of 22} \]

\[ \text{lg := groebner(lp)$lextripack} \]

\[ \text{--R} \]

\[ \text{--R (13)} \]

\[ \text{[a + b + c + d + e + f,} \]

\[ \text{--R} \]

\[ \text{--R 2} \]

\[ 3968379498283200b + 15873517993132800f b + 3968379498283200d} \]

\[ \text{--R +} \]

\[ \text{--R 3 5 4 4} \]

\[ 15873517993132800f d + 3968379498283200f e - 15873517993132800f e \]

\[ \text{--R +} \]

\[ \text{--R 5 3 6 2} \]

\[ 238102769896999200f e + (206355733910726400f + 230166010900425600)e \]

\[ \text{--R +} \]

\[ \text{--R 43 37} \]

\[ - 729705987316687f + 1863667496867205421f} \]

\[ \text{--R +} \]

\[ \text{--R 31 25} \]

\[ 291674853771731104461f + 365285994691106921745f} \]

\[ \text{--R +} \]

\[ \text{--R 19 13} \]
\[
\begin{align*}
& -R 549961185828911895f - 3650484040387668439269f \\
& + 7 \\
& -R 292382820431504027669f - 2271898467631865497f \\
& * e \\
& + 44 \quad 38 \\
& - R 398812642545399f + 10187423878429609997f \\
& + 32 \quad 26 \\
& - R 159437752342314053637f + 1994739308439916238065f \\
& + 20 \quad 14 \\
& - R 1596840088052642815f - 1993494118301162145413f \\
& + 8 \quad 2 \\
& - R 1596049742289689815053f - 114881712330159667449f \\
& , \\
& , \\
& (23810276989699200c - 23810276989699200f)b + 23810276989699200c \\
& + 2 \\
& - R 71430830969097600f c - 23810276989699200d - 9524110795896800f d \\
& + 3 5 \quad 4 4 \\
& - R 55557312975964800f e + 174608697924460800f e - 174608697924460800f e \\
& + 6 \quad 2 \\
& - R (-2428648252949318400f - 2611193709870345600e)
& + 43 \quad 37 \\
& - R 8305444561289527f - 21212087151945459641f \\
& + 31 \quad 25 \\
& - R 3319815883093451385381f - 415769164626165713645f \\
& + 19 \quad 13 \\
& - R 6072721607510764095f + 4154986709036460221649f \\
& + 7 \\
& - R 3327761311138587096749f + 25885340608290841637f \\
& * e \\
& + 44 \quad 38 \\
& 45815897629010329f - 117013765582151891207f \\
& + 32 \quad 26
\end{align*}
\]
\begin{verbatim}
--R  6  2
--R  +
--R  43  37
--R  +
--R  31  25
--R  +
--R  786029984751110f - 2007519008182245250f
--R  +
--R  314892372799176495730f + 240938651514668530f
--R  *
--R  e
--R  +
--R  44  38  32
--R  4177638546747827f - 10669685294602576381f - 1669852980419949524601f
--R  +
--R  26  20
--R  +
--R  14  8
--R  2087864026859015573349f + 1671496085945199577969f
--R  +
--R  2
--R  11940257226216280177f
--R  ,
--R  6  2  5
--R  (11905138494849600f - 11905138494849600) b - 15873517993132800f e
--R  +
--R  3  4  3
--R  39683794982832000f e - 39683794982832000f e
--R  +
--R  11  5  2
--R  ( - 686529653202993600f - 60716206327329600f ) e
--R  +
--R  42  36  30
--R  65144531306704f - 166381280901088652f - 26033434502470283473f
--R  +
--R  24  18
--R  +
--R  12  6
--R  32220085033691389548f + 25526177666070529808f + 138603268355749244
--R  *
--R  e
--R  +
\end{verbatim}
CHAPTER 13. CHAPTER L

\[
\begin{align*}
&+ 43 \quad 37 \quad 31 \\
&167620036074811f \quad - 428102417974791473f \quad - 66997243801231679313f \\
&+ 25 \quad 19 \\
&- 83426716722148750485f \quad + 203673895369980765f \\
&+ 13 \quad 7 \\
&83523056326010432457f \quad + 66995789640238066937f \quad + 478592855549587901f \\
, \\
3 \quad 2 \quad 2 \quad 45 \\
&801692827936c \quad + 2405078483808fc \quad - 2405078483808fc \quad - 13752945467f \\
&+ 39 \quad 33 \quad 27 \\
&35125117815561f \quad + 5496946957826433f \quad + 6834659447749117f \\
&+ 21 \quad 15 \quad 9 \\
&44484880462461f \quad - 6873406230093057f \quad - 5450844938762633f \\
&+ 3 \\
&121658604457f \\
, \\
2 \\
&(23810276989699200d \quad - 23810276989699200f)c \quad + 23810276989699200d \\
&+ 3 \quad 5 \quad 4 \quad 4 \\
&71430830969097600f \quad d \quad + 7936758965666400f \quad e \quad - 31747035986265600f \quad e \\
&+ 5 \quad 3 \quad 6 \quad 2 \\
&31747035986265600f \quad e \quad + (404774708824886400f \quad + 396837949828320000)e \\
&+ 43 \quad 37 \\
&- 124737229446701f \quad + 3185785654596621203f \\
&+ 31 \quad 25 \\
&498594866849974751463f \quad + 624542545845791047936f \\
&+ 19 \quad 13 \\
&931085755769682885f \quad - 624150663582417063387f \\
&+ 7 \\
&- 499881859388360475647f \quad - 3926885313819527351f \\
\end{align*}
\]
--R 2808383522593986603543f + 3513624142354807530135f
--R +
--R 20 14
--R 2860757006705537685f - 3511356735642190737267f
--R +
--R 8 2
--R - 2811332494697103819887f - 20315011631522847311f
--R +
(7936758996566400e - 7936758996566400f)c
--R +
--R 43 37 31
--R - 4418748183673f + 11285568707456559f + 1765998617294451019f
--R +
--R +
--R 25 19
--R 2173749283622606155f - 55788292195402895f
--R +
--R 13 7
--R - 221529142178292951f - 171814266534730851f + 30256569458230237f
--R *
--R e
--R +
--R 44 38 32
--R 4418748183673f - 11285568707456559f - 1765998617294451019f
--R +
--R 26 20 14
--R - 2173749283622606155f + 55788292195402895f + 221529142178292951f
--R +
--R +
--R 8 2
--R 1718142665347430851f - 30256569458230237f
--R +
--R 6 43
--R (72152354514240f - 72152354514240)c + 40950859499f
--R +
--R 37 31 25
--R - 104588980990367f - 16367227395575307f - 20268523416527355f
--R +
--R +
--R 19 13 7
--R 442205002259535f + 20576059935789063f + 15997133796970563f
--R +
--R - 275099892785581f
--R +
--R 3 2 2
--R 1984189749141600d + 5952569247424800f d - 5952569247424800f d
--R +
--R 4 5 5 4 3
--R - 3968379498283200f e + 15873517993132800f e + 17857707742274400e
--R +
\[
\begin{align*}
\text{--R} & \quad 7 \quad 2 \\
\text{--R} & \quad (-148814231185620000f - 16270355942961200f)e \\
\text{--R} & \quad + \\
\text{--R} & \quad 44 \quad 38 \\
\text{--R} & \quad -390000914678878f + 996062704593756434f \\
\text{--R} & \quad + \\
\text{--R} & \quad 32 \quad 26 \\
\text{--R} & \quad 155886323972034823914f + 194745956143985421330f \\
\text{--R} & \quad + \\
\text{--R} & \quad 20 \quad 14 \\
\text{--R} & \quad 620577595574430f - 19459651265329968786f \\
\text{--R} & \quad + \\
\text{--R} & \quad 8 \quad 2 \\
\text{--R} & \quad -155796897940756922666f - 1036375759077320978f \\
\text{--R} & \quad * \\
\text{--R} & \quad e \\
\text{--R} & \quad + \\
\text{--R} & \quad 45 \quad 39 \quad 33 \\
\text{--R} & \quad -374998630035991f + 957747106595453993f + 14988915566764891693f \\
\text{--R} & \quad + \\
\text{--R} & \quad 27 \quad 21 \\
\text{--R} & \quad 187154171443494641685f - 127129015426348065f \\
\text{--R} & \quad + \\
\text{--R} & \quad 15 \quad 9 \quad 3 \\
\text{--R} & \quad -187241533243115040417f - 149719983567976534037f - 836654081239648061f \\
\text{--R} & \quad , \\
\text{--R} & \quad 3 \quad 5 \\
\text{--R} & \quad (5952569247424800e - 5952569247424800f)d - 3968379498283200f \ e \\
\text{--R} & \quad + \\
\text{--R} & \quad 4 \quad 4 \quad 5 \quad 3 \\
\text{--R} & \quad 9920948745708000f \ e - 3968379498283200f \ e \\
\text{--R} & \quad + \\
\text{--R} & \quad 6 \quad 2 \\
\text{--R} & \quad (-148814231185620000f - 15079842093476160)e \\
\text{--R} & \quad + \\
\text{--R} & \quad 43 \quad 37 \\
\text{--R} & \quad 492558110242553f - 1257992359608075499f \\
\text{--R} & \quad + \\
\text{--R} & \quad 31 \quad 25 \\
\text{--R} & \quad -19688309453968513959f - 246562115745735428055f \\
\text{--R} & \quad + \\
\text{--R} & \quad 19 \quad 13 \\
\text{--R} & \quad -325698701993885505f + 246417769883651808111f \\
\text{--R} & \quad + \\
\text{--R} & \quad 7 \\
\text{--R} & \quad 197327352068200652911f + 1523373796389332143f \\
\text{--R} & \quad * \\
\text{--R} & \quad e \\
\text{--R} & \quad +
\end{align*}
\]
-- R 4 4 3 8 3 2
-- R 4 2 6 9 5 4 2 1 3 0 6 0 8 f - 1 0 7 1 0 2 0 4 5 6 4 2 6 9 1 3 5 7 8 f
-- R +
-- R 2 6 2 0
-- R 1 3 9 7 8 9 1 6 9 6 9 2 0 4 1 2 4 0 0 6 0 f - 8 5 2 7 4 6 7 5 0 9 1 0 7 5 0 1 1 0 f
-- R +
-- R 1 4 8
-- R 1 3 9 1 0 5 1 0 1 9 7 1 8 7 8 4 0 1 3 1 2 f + 1 0 7 1 9 0 0 2 8 9 7 5 8 7 1 2 9 8 4 7 6 2 f
-- R +
-- R 2
-- R 7 5 5 2 3 9 0 7 2 0 7 2 7 2 7 7 5 6 f
-- R ,
-- R 6 2 5
-- R (1 1 9 0 5 1 3 8 4 9 4 8 4 9 6 0 0 f - 1 1 9 0 5 1 3 8 4 9 4 8 4 9 6 0 0 ) d - 7 9 3 6 7 5 8 9 9 6 5 6 4 0 0 f e
-- R +
-- R 3 4 4 3
-- R 3 1 7 4 7 0 3 5 9 8 6 2 6 5 6 0 0 f e - 3 1 7 4 7 0 3 5 9 8 6 2 6 5 6 0 0 f e
-- R +
-- R 1 1 5 2
-- R (- 4 2 0 6 4 8 2 2 6 8 1 8 0 1 9 2 0 0 f - 4 0 4 7 7 4 7 0 8 8 2 4 8 8 6 4 0 0 f ) e
-- R +
-- R 4 2 3 6 3 0
-- R 1 5 3 6 1 8 7 6 0 0 8 8 9 f - 3 9 1 6 9 7 3 9 5 6 5 1 6 1 1 0 7 f - 6 1 2 7 1 7 6 1 2 7 4 8 9 6 9 0 8 2 7 f
-- R +
-- R 2 4 1 8
-- R - 7 2 1 7 7 0 8 7 4 2 3 1 0 5 0 9 6 1 5 f + 5 3 8 6 2 8 4 8 3 8 9 0 7 2 2 7 3 5 f
-- R +
-- R 1 2 6
-- R 7 5 0 6 8 0 4 3 5 3 8 4 3 5 0 7 6 4 3 f + 5 8 8 6 1 6 0 7 6 9 7 8 2 6 0 7 2 0 3 f + 6 3 5 7 6 1 0 8 3 9 6 5 3 5 8 7 9
-- R *
-- R e
-- R +
-- R 4 3 3 7 3 1
-- R 7 1 7 3 7 7 8 1 7 7 7 0 6 6 f - 1 8 3 2 1 8 8 5 6 2 0 7 5 5 7 9 3 8 f - 2 8 6 7 2 8 7 4 2 7 1 1 3 2 2 7 6 0 7 8 f
-- R +
-- R 2 5 1 9
-- R - 3 5 6 2 5 2 2 3 6 8 6 9 3 9 8 1 2 0 1 0 f + 1 6 4 8 3 1 3 3 9 6 3 4 0 8 4 3 9 0 f
-- R +
-- R 1 3 7
-- R 3 5 7 2 4 1 6 0 4 2 3 0 7 3 0 5 2 6 4 2 f + 2 8 6 2 7 0 2 2 5 7 8 6 6 4 9 1 0 6 2 2 f + 1 8 7 4 5 9 9 8 7 0 2 9 6 8 0 5 0 6 f
-- R ,
-- R 6 5 2 4
-- R 1 3 2 2 7 9 3 1 6 6 0 9 4 4 0 0 e - 3 9 6 8 3 7 9 4 9 8 2 8 3 2 0 0 f e + 3 9 6 8 3 7 9 4 9 8 2 8 3 2 0 0 f e
-- R +
-- R 3 3
-- R - 5 2 9 1 1 7 2 6 6 4 3 7 7 6 0 0 f e
-- R +
-- R 1 0 4 2
--R  ( - 230166010900425600f - 226197631402142400f ) e
--R  +
--R  47  41
--R  - 152375364610443885f + 389166626064854890415f
--R  +
--R  35  29
--R  60906097841360558987335f + 76167367934608798697275f
--R  +
--R  23  17
--R  27855066785995181125f - 76144952817052723145495f
--R  +
--R  11  5
--R  - 60933629892463517546975f - 411415071682002547795f
--R  *
--R  e
--R  +
--R  42  36  30
--R  - 209493533143822f + 535045979490560586f + 8373794796497353146f
--R  +
--R  24  18
--R  104889507084213371570f + 167117997269207870f
--R  +
--R  12  6
--R  - 104793725781390615514f - 83842685189903180394f - 569978796672974242
--R  ,
--R  6  3
--R  (25438330117200f + 25438330117200)e
--R  +
--R  7  2
--R  (76314990351600f + 76314990351600f)e
--R  +
--R  44  38  32
--R  - 1594966552735f + 4073543370415745f + 637527159231148925f
--R  +
--R  26  20  14
--R  797521176113606525f + 530440941097175f - 797160527306433145f
--R  +
--R  8  2
--R  - 638132320196044966f - 4510507167940726f
--R  *
--R  e
--R  +
--R  45  39  33
--R  - 6036376800443f + 15416903421476909f + 2412807646192304449f
--R  +
--R  27  21  15
--R  3017679923028013705f + 1422320037411955f - 3016560402417843941f
--R  +
--R  9  3
--R - 2414249368183033161f - 16561862361763873f
--R ,
--R
--R 12 2
--R (1387545279120f - 1387545279120)e
--R +
--R 31 43 37
--R 4321823003f - 11037922310209f - 1727510711947989f
--R +
--R 19 25
--R - 2165150991154425f - 5114342560755 + 2162682824948601f
--R +
--R
--R 7
--R 1732620732685741f + 13506088516033f
--R *
--R e
--R +
--R 32 44 38
--R 24177661775f - 61749727185325f - 9664106795754225f
--R +
--R 20 26
--R - 12090487758628245f - 8787672733575 + 12083693383005045f
--R +
--R 2 8
--R 9672870290826025f + 68544102808525f
--R ,
--R 6 6 5 2 4 3 3 4 2 5
--R {f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,}
--R 2 5 3 4 4 3 5 2
--R 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
--R 2 5 3 4 4 3 5 2
--R 2 5 3 4 4 3 5 2
--R a - f e + 3f e - 3f e + 4f e + 3e - 3f}
--R ,
--R 6 6 5 2 4 3 3 4 2 5
--R {f - 1,e - f,d - f,c + 4f c + f ,,(c - f)b - f c - 5f ,a + b + c + 3f},
--R 6 2 2
--R {f - 1,e - f,d - f,c - f,b + 4f b + f ,a + b + 4f},

Type: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))

--- 13

--E 13

--S 14 of 22

lexTriangular(1g, false)$lextripack

--R
--R
--R (14)
--R [
--R 6 6 5 2 4 3 3 4 2 5
--R {f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,}
--R 2 5 3 4 4 3 5 2
--R 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
--R 2 5 3 4 4 3 5 2
--R 2 5 3 4 4 3 5 2
--R a - f e + 3f e - 3f e + 4f e + 3e - 3f}
--R ,
--R 6 6 5 2 4 3 3 4 2 5
--R {f - 1,e - f,d - f,c + 4f c + f ,,(c - f)b - f c - 5f ,a + b + c + 3f},
--R 6 2 2
--R {f - 1,e - f,d - f,c - f,b + 4f b + f ,a + b + 4f},

Type: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))

--- 13

--E 13

--S 14 of 22

lexTriangular(1g, false)$lextripack

--R
--R
--R (14)
--R [
--R 6 6 5 2 4 3 3 4 2 5
--R {f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,}
--R 2 5 3 4 4 3 5 2
--R 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
--R 2 5 3 4 4 3 5 2
--R 2 5 3 4 4 3 5 2
--R a - f e + 3f e - 3f e + 4f e + 3e - 3f}
--R ,
--R 6 6 5 2 4 3 3 4 2 5
--R {f - 1,e - f,d - f,c + 4f c + f ,,(c - f)b - f c - 5f ,a + b + c + 3f},
--R 6 2 2
--R {f - 1,e - f,d - f,c - f,b + 4f b + f ,a + b + 4f},
\[ R \{ f - 1, e - f, d + 4f, d + f, (d - f)c - f, d - 5f, b - f, a + c + d + 3f \}, \]
\[ R \{ 36, 30, 24, 18, 12, 6 \}
\[ R \{ f - 255f, - 399709f, - 502276f, - 399709f, - 2554f + 1 \}, \]
\[ R \{ 12, 2 \}
\[ (161718564f - 161718564)e, \]
\[ R \{ 31, 25, 19, 13 \}
\[ - 504205f + 1287737951f + 201539391380f + 253982817368f \]
\[ + 7 \]
\[ 201940704685f + 1574134601f \]
\[ R \{ 6, 2, 5, 3, 4 \}
\[ 3625 \]
\[ (693772639560f - 693772639560)d - 462515093040f e + 1850060372160f e \]
\[ - 1850060372160f e + (- 24513299931120f - 23588269745040f )e \]
\[ R \{ 6, 2 \}
\[ 3024 \]
\[ - 890810428f + 2275181044754f + 355937263869776f \]
\[ + 8 \]
\[ 112737589345f + 7988820725f \]
\[ R \{ 6, 31, 25 \}
\[ (12614047992f - 12614047992)c - 7246825f + 18508536599f \]
\[ - 18508536599f + 10634395752169f + 1664161760192806f \]
\[ + 13 \]
\[ 2079424391370694f + 1668153650635921f + 10924274392693f \]
\[ R \{ 6, 31, 25 \}
\[ 2896249516034f + 3581539649666f + 2796477571739f - 48094301893f \].
PACKAGE LEXTRIPK LEXTRIANGULARPACKAGE

--R ,
--R [ 6 5 3 4 2 3 4 3 2 5 ]
--R (693772639560f - 693772639560) b - 925030186080f e + 2312575465200f e
--R +
--R 11 5 2
--R - 2312575465200f e + (- 40007555547960f - 35382404617560f ) e
--R +
--R 24 20 18
--R - 3781280823f + 9657492291789f + 1511158913397906f e + 2312575465200f e
--R +
--R 12 6
--R 6 2
--R 2 5 3 4 4 3 5 2
--R a + b + c + d + e + f}
--R ,
--R 6 2 2 2
--R {f - 1, e + 4f e + f , (e - f)d - f e - 5f , c - f , b - f, a + d + e + 3f}
--R Type: List(RegularChain(Integer,[a,b,c,d,e,f]))
--E 14

S 15 of 22
lts := lexTriangular(lg, true) $ lextripack
--R
--R (15)
--R [ 5 4 3 2 6 6 5 2 4 3 3 4 2 5 ]
--R {f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,}
--R 2 5 3 4 4 3 5 2
--R 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
--R 2 5 3 4 4 3 5 2
--R 3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f,
--R 2 5 3 4 4 3 5 2
--R a - f e + 3f e - 3f e + 4f e + 3e - 3f}
--R ,
--R 6 2 2
--R {f - 1, e - f,d - f,c + 4f c + f , b + c + 4f,a - f},
--R 6 2 2
--R {f - 1, e - f, d - f, c - f , b + 4f b + f , a + b + 4f},
--R 6 2 2
--R {f - 1, e - f, d + 4f d + f , c + d + 4f, b - f, a - f},
\[
\begin{align*}
&\{ f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \\
&2 \\
&1387545279120e \\
&+ \\
&4321823003f - 11037922310209f - 1727506390124986f \\
&+ \\
&- 2176188913464634f - 1732620732685741f - 13506088516033f \\
&* \\
&e \\
&+ \\
&32177661775f - 61749727185325f - 9664082618092450f \\
&+ \\
&- 12152237485813570f - 9672870290826025f - 68544102808525f \\
&, \\
&1387545279120d \\
&+ \\
&30128983050f + 2883434331830f + 451234998755840f \\
&+ \\
&125624691685760f + 447129055314890f - 165557857270 \\
&* \\
&e \\
&+ \\
&311816935351f + 4640452214013f + 726247129626942f \\
&+ \\
&912871801716798f + 726583262666877f + 4909358645961f \\
&, \\
&1387545279120c + 778171189f - 1987468196267f - 310993556954378f \\
&+ \\
&1383262822316802f - 300335488637543f + 5289595037041f \\
&, \\
&1387545279120b \\
&+ \\
&1128983050f - 2883434331830f - 451234998755840f 
\end{align*}
\]
\begin{verbatim}
--R + 12 6
--R - 562426491685760f - 4471290553148890f + 165557857270
--R * e
--R + 31 25 19
--R - 3283058841f + 8384938292463f + 1312252817452422f
--R + 13 7
--R 1646579934064638f + 1306372958656407f + 4694680112151f
--R ,
--R --R 31 25
--R 1387545279120a + 1387545279120e + 4321823003f - 11037922310209f
--R + 19 13 7
--R - 1727506390124986f - 2176188913464634f - 1732620732685741f
--R +
--R - 13506088516033f
--R ,
--R --R 6 2 2
--R {f - 1, e + 4f e + f, d + e + 4f, c - f, b - f, a - f}
--R Type: List(RegularChain(Integer,[a,b,c,d,e,f]))
\end{verbatim}
\begin{align*}
& R_2^5 \cdot 34 \cdot 43 \cdot 52 \\
& a - f + 3f - e - 3f e + 4f e + 3e - 3f \\
& ,
\end{align*}

\begin{align*}
& R_6 \cdot 2^2 \\
& \{f - 1, e - f, d - f, c + 4f, c + f, b + c + 4f, a - f\}, \\
& \{f - 1, e - f, d - f, c - f, b + 4f b + f, a + b + 4f\}, \\
& \{f - 1, e - f, d + 4f d + f, c + d + 4f, b - f, a - f\},
\end{align*}

\begin{align*}
& R_36 \cdot 30 \cdot 24 \cdot 18 \cdot 12 \cdot 6 \\
& \{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \\
& R_2 \cdot 2 \\
& 1387545279120e \\
& + \\
& R_31 \cdot 25 \cdot 19 \\
& 4321823003f - 11037922310209f - 1727506390124986f \\
& + \\
& R_13 \cdot 7 \\
& - 2176188913464634f - 1732620732685741f - 13506088516033f \\
& * \\
& R_e \\
& + \\
& R_32 \cdot 26 \cdot 20 \\
& 24177661775f - 61749727185325f - 9664082618092450f \\
& + \\
& R_14 \cdot 8 \cdot 2 \\
& - 12152237485813570f - 9672870290826025f - 68544102808525f \\
& , \\
& R_1387545279120d \\
& + \\
& R_30 \cdot 24 \cdot 18 \\
& - 1128983060f + 2883434331830f + 451234998755840f \\
& + \\
& R_12 \cdot 6 \\
& 562426491685760f + 447129055314890f - 165557857270 \\
& * \\
& R_e \\
& + \\
& R_31 \cdot 25 \cdot 19 \\
& - 1816935351f + 4640452214013f + 726247129626942f \\
& + \\
& R_912871801716798f + 726583262666877f + 4909358645961f \\
& , \\
& R_1387545279120c + 778171189f - 1987468196267f - 310993556954378f
```
--R +
--R 13
--R 7
--R - 383262822316802f - 300335488637543f + 5289595037041f
--R ,
--R 1387545279120b
--R +
--R 30
--R 24
--R 18
--R 12
--R 6
--R - 56246491685760f - 447129055314890f + 165557857270
--R *
--R e
--R +
--R 31
--R 25
--R 19
--R - 3283058841f + 8384938292463f + 13122528174522f
--R +
--R 13
--R 7
--R 1646579934064638f + 130637958656407f + 4694680112151f
--R ,
--R 31
--R 25
--R 1387545279120a + 1387545279120e + 4321823003f - 11037922310209f
--R +
--R 19
--R 13
--R - 1727506390124986f - 217618913464634f - 1732620732685741f
--R +
--R - 13506088516033f
--R }
--R ,
--R 6
--R 2
--R 2
--R f - 1,e + 4f e + f,d + e + 4f,c - f,b - f,a - f]
--R Type: List(SquareFreeRegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([a,b,c,d,e,f])),OrderedVariableList([a,b,c,d,e,f])),NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))))
--E 17

--S 18 of 22
reduce(+,[degree(ts) for ts in lts])
--R
--R
--R (18) 156
--R Type: PositiveInteger
--E 18

--S 19 of 22
ls2 : List Symbol := concat(ls,new()$Symbol)
--R
--R
--R (19) [a,b,c,d,e,f,%A]
--R Type: List(Symbol)
```
zdpack := ZDSOLVE(R,ls,ls2)
(20) ZeroDimensionalSolvePackage(Integer, [a,b,c,d,e,f], [a,b,c,d,e,f,%A])
Type: Domain

concat [univariateSolve(ts)$zdpack for ts in lts]
(21) 
[4 2 
complexRoots= ? - 13? + 49, 
coordinates = 
3 3 3 3 
[7a + %A - 6%A, 21b + %A + %A, 21c - 2%A + 19%A, 7d - %A + 6%A, 
3 3 
21e - %A - %A, 21f + 2%A - 19%A] 
] 
, 
4 2 
[complexRoots= ? + 11? + 49, 
coordinates = 
3 3 
[35a + 3%A + 19%A, 35b + %A + 18%A, 35c - 2%A - %A, 
3 3 
35d - 3%A - 19%A, 35e - %A - 18%A, 35f + 2%A + %A] 
] 
, 
[ 
complexRoots = 
8 7 6 5 4 3 2 
, 
coordinates = 
[ 
7 6 5 4 
43054532a + 33782%A - 546673%A + 3127348%A - 6927123%A 
+ 
3 2 
]
complexRoots =

coordinates =

[]
```
CHAPTER 13. CHAPTER L

--R + 3 2
--R 4365212%A + 25086957%A + 39582814%A + 107313172
--R ,
--R
--R 7 6 5 4
--R 43054532b - 33782%A - 546673%A - 3127348%A - 6927123%A
--R +
--R 3 2
--R - 4365212%A - 25086957%A - 39582814%A - 107313172
--R ,
--R
--R 7 6 5 4
--R 21527266c - 22306%A - 263139%A - 1166076%A - 1821805%A
--R +
--R 3 2
--R - 2892788%A - 10322663%A - 9026596%A - 12950740
--R ,
--R
--R 7 6 5 4
--R 43054532d + 22306%A + 263139%A + 1166076%A + 1821805%A
--R +
--R 3 2
--R 2892788%A + 10322663%A + 30553862%A + 12950740
--R ,
--R
--R 7 6 5 4
--R 43054532e - 22306%A - 263139%A - 1166076%A - 1821805%A
--R +
--R 3 2
--R - 2892788%A - 10322663%A - 30553862%A - 12950740
--R ,
--R
--R 7 6 5 4
--R 21527266f + 22306%A + 263139%A + 1166076%A + 1821805%A
--R +
--R 3 2
--R 2892788%A + 10322663%A + 9026596%A + 12950740
--R ]
--R }
--R ,
--R
--R 4 2
--R [complexRoots= ? - ? + 1,
--R 3 3 3 3
--R coordinates= [a - %A,b + %A - %A,c + %A,d + %A,e - %A + %A,f - %A ]]
--R ,
--R
--R 8 6 4 2
--R [complexRoots= ? + 4? + 12? + 16? + 4,
```
coordinates =

\[
\begin{bmatrix}
7 & 5 & 3 & 7 & 5 & 3 \\
4a - 2%A - 7%A - 20%A - 22%A, & 4b + 2%A + 7%A + 20%A + 22%A, \\
7 & 5 & 3 & 7 & 5 & 3 \\
4c + %A + 3%A + 10%A + 10%A, & 4d + %A + 3%A + 10%A + 6%A, \\
7 & 5 & 3 & 7 & 5 & 3 \\
4e - %A - 3%A - 10%A - 6%A, & 4f - %A - 3%A - 10%A - 10%A \\
\end{bmatrix}
\]

complexRoots = \[ \pm 6? + 30? + 36? + 36, \]

coordinates =

\[
\begin{bmatrix}
3 & 2 & 3 & 2 \\
30a - %A - 5%A - 30%A - 6, & 6b + %A + 5%A + 24%A + 6, \\
3 & 2 & 3 & 2 \\
30c - %A - 5%A + 6, & 30d - %A - 5%A - 30%A - 6, \\
3 & 2 & 3 & 2 \\
30e - %A - 5%A - 30%A - 6, & 30f - %A + 5%A - 30%A - 6 \\
\end{bmatrix}
\]

complexRoots = \[ \pm 6? - 30? - 36? + 36, \]

coordinates =

\[
\begin{bmatrix}
3 & 2 & 3 & 2 \\
30a - %A + 5%A - 30%A + 6, & 6b + %A - 5%A + 24%A - 6, \\
3 & 2 & 3 & 2 \\
30c - %A + 5%A + 6, & 30d - %A + 5%A - 30%A + 6, \\
3 & 2 & 3 & 2 \\
30e - %A + 5%A + 30%A + 6, & 30f - %A + 5%A - 30%A + 6 \\
\end{bmatrix}
\]

complexRoots = \[ \pm 6? + 6, \]

coordinates = \[a + 1, b - %A - 5, c + %A + 1, d + 1, e + 1, f + 1] \]

complexRoots = \[ \pm 6? - 6, \]

coordinates = \[a - 1, b - %A + 5, c + %A - 1, d - 1, e - 1, f - 1] \]

complexRoots = \[ \pm 6? + 30? + 36? + 36, \]
coordinates =
3 2 3 2
[6a + %A + 5%A + 24%A + 6, 30b + %A + 5%A + 6,
3 2 3 2
30c - %A - 5%A - 30%A - 6, 30d - %A - 5%A - 30%A - 6,
3 2 3 2
30e - %A - 5%A - 30%A - 6, 30f - %A - 5%A - 30%A - 6]
,
,
4 3 2
[complexRoots= ? - 6? + 30? - 36? + 36,

coordinates =
3 2 3 2
[6a + %A - 5%A + 24%A - 6, 30b - %A + 5%A + 6,
3 2 3 2
30c - %A + 5%A - 30%A + 6, 30d - %A + 5%A - 30%A + 6,
3 2 3 2
30e - %A + 5%A - 30%A + 6, 30f - %A + 5%A - 30%A + 6]
,
,
2
[complexRoots= ? + 6? + 6,
coordinates= [a - %A - 5,b + %A + 1,c + 1,d + 1,e + 1,f + 1]]
,
,
2
[complexRoots= ? - 6? + 6,
coordinates= [a - %A + 5,b + %A - 1,c - 1,d - 1,e - 1,f - 1]]
,
,
4 3 2
[complexRoots= ? + 6? + 30? + 36? + 36,
coordinates =
3 2 3 2
[30a - %A - 5%A - 30%A - 6, 30b - %A - 5%A - 30%A - 6,
3 2 3 2
6c + %A + 5%A + 24%A + 6, 30d - %A - 5%A - 6,
3 2 3 2
30e - %A - 5%A - 30%A - 6, 30f - %A - 5%A - 30%A - 6]
,
,
4 3 2
[complexRoots= ? - 6? + 30? - 36? + 36,
coordinates =
\[
\begin{align*}
\text{--R} & \quad 3 & 2 & 3 & 2 \\
\text{--R} & \quad [30a - \%A + 5\%A - 30\%A + 6, 30b - \%A + 5\%A - 30\%A + 6, \\
\text{--R} & \quad 3 & 2 & 3 & 2 \\
\text{--R} & \quad 6c + \%A - 5\%A + 24\%A - 6, 30d - \%A + 5\%A + 6, \\
\text{--R} & \quad 3 & 2 & 3 & 2 \\
\text{--R} & \quad 30e - \%A + 5\%A - 30\%A + 6, 30f - \%A + 5\%A - 30\%A + 6] \\
\text{--R} & \quad , \\
\text{--R} & \quad , \\
\text{--R} & \quad 2 \\
\text{--R} & \quad \text{[complexRoots} = ? + 6? + 6, \\
\text{--R} & \quad \text{coordinates} = [a + 1,b + 1,c - \%A - 5,d + \%A + 1,e + 1,f + 1] \\
\text{--R} & \quad , \\
\text{--R} & \quad , \\
\text{--R} & \quad 2 \\
\text{--R} & \quad \text{[complexRoots} = ? - 6? + 6, \\
\text{--R} & \quad \text{coordinates} = [a - 1,b - 1,c - \%A + 5,d + \%A - 1,e - 1,f - 1] \\
\text{--R} & \quad , \\
\text{--R} & \quad , \\
\text{--R} & \quad 8 & 7 & 6 & 5 & 4 & 2 \\
\text{--R} & \quad \text{[complexRoots} = ? + 6? + 16? + 24? + 18? - 8? + 4, \\
\text{--R} & \quad \text{coordinates} = \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad [2a + 2\%A + 9\%A + 18\%A + 19\%A + 4\%A - 10\%A - 2\%A + 4, \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad 2b + 2\%A + 9\%A + 18\%A + 19\%A + 4\%A - 10\%A - 4\%A + 4, \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 \\
\text{--R} & \quad 2c - \%A - 4\%A - 8\%A - 9\%A - 4\%A - 2\%A - 4, \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 \\
\text{--R} & \quad 2d + \%A + 4\%A + 8\%A + 9\%A + 4\%A + 2\%A + 4, \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad 2e - 2\%A - 9\%A - 18\%A - 19\%A - 4\%A + 10\%A + 4\%A - 4, \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad 2f - 2\%A - 9\%A - 18\%A - 19\%A - 4\%A + 10\%A + 2\%A - 4] \\
\text{--R} & \quad , \\
\text{--R} & \quad , \\
\text{--R} & \quad [ \\
\text{--R} & \quad \text{complexRoots} = \\
\text{--R} & \quad 8 & 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad , \\
\text{--R} & \quad , \\
\text{--R} & \quad \text{coordinates} = \\
\text{--R} & \quad [ \\
\text{--R} & \quad 7 & 6 & 5 & 4 & 3 & 2 \\
\text{--R} & \quad 1408a - 19\%A - 200\%A - 912\%A - 2216\%A - 4544\%A - 6784\%A \\
\text{--R} & \quad + \\
\text{--R} & \quad - 6976\%A - 1792
\end{align*}
\]
CHAPTER 13. CHAPTER L

--R
--R
--R 7 6 5 4 3 2
--R
--R 1408b - 37%A - 408%A - 1952%A - 5024%A - 10368%A - 16768%A
--R +
--R - 17920%A - 5120
--R ,
--R
--R 7 6 5 4 3 2
--R 1408c + 37%A + 408%A + 1952%A + 5024%A + 10368%A + 16768%A
--R +
--R 17920%A + 5120
--R ,
--R
--R 7 6 5 4 3 2
--R 1408d + 19%A + 200%A + 912%A + 2216%A + 4544%A + 6784%A
--R +
--R 6976%A + 1792
--R ,
--R 2e + %A, 2f - %A
--R ]
--R ,
--R
--R 8 6 4 2
--R [complexRoots= ? + 4? + 12? + 16? + 4,
--R coordinates =
--R
--R 7 5 3
--R [4a - %A - 3%A - 10%A - 6%A, 4b - %A - 3%A - 10%A - 10%A,
--R 7 5 3
--R 4c - 2%A - 7%A - 20%A - 22%A, 4d + 2%A + 7%A + 20%A + 22%A,
--R 7 5 3
--R 4e + %A + 3%A + 10%A + 10%A, 4f + %A + 3%A + 10%A + 6%A
--R ]
--R ,
--R
--R 8 6 4 2
--R [complexRoots= ? + 16? - 96? + 256? + 256,
--R coordinates =
--R
--R 7 5 3
--R [512a - %A - 12%A + 176%A - 448%A,
--R 7 5 3
--R 128b - %A - 16%A + 96%A - 256%A,
--R 7 5 3
--R 128c + %A + 16%A - 96%A + 256%A,
--R 7 5 3
--R 512d + %A + 12%A - 176%A + 448%A, 2e + %A, 2f - %A
--R ]
--R ,
--R [
--R  complexRoots =
--R     8  7  6  5  4  3  2
--R ,
--R  coordinates =
--R  [
--R      7  6  5  4  3  2
--R     1408a - 19%A + 200%A - 912%A + 2216%A - 4544%A + 6784%A
--R     +
--R     - 6976%A + 1792
--R ,
--R      7  6  5  4  3  2
--R     1408b - 37%A + 408%A - 1952%A + 5024%A - 10368%A + 16768%A
--R     +
--R     - 17920%A + 5120
--R ,
--R      7  6  5  4  3  2
--R     1408c + 37%A - 408%A + 1952%A - 5024%A + 10368%A - 16768%A
--R     +
--R     17920%A - 5120
--R ,
--R      7  6  5  4  3  2
--R     1408d + 19%A - 200%A + 912%A - 2216%A + 4544%A - 6784%A
--R     +
--R     6976%A - 1792
--R ,
--R 2e + %A, 2f - %A]
--R ]
--R ,
--R  coordinates =
--R     7  6  5  4  3  2
--R     [2a + 2%A - 9%A + 18%A - 19%A + 4%A + 10%A - 2%A - 4,
--R     7  6  5  4  3  2
--R     2b + 2%A - 9%A + 18%A - 19%A + 4%A + 10%A - 4%A - 4,
--R     7  6  5  4  3
--R     2c - %A + 4%A - 8%A + 9%A - 4%A - 2%A + 4,
--R     7  6  5  4  3
--R     2d + %A - 4%A + 8%A - 9%A + 4%A + 2%A - 4,
--R     7  6  5  4  3
--R     2e - 2%A + 9%A - 18%A + 19%A - 4%A - 10%A + 4%A + 4,
"complexRoots= ? + 12? + 144,
coordinates = 
[12a - %A - 12, 12b - %A - 12, 12c - %A - 12, 12d - %A - 12, 
  6e + %A + 3%A + 12, 6f + %A - 3%A + 12]
"

"complexRoots= ? + 6? + 30? + 36? + 36,
coordinates = 
[6a - %A - 5%A - 24%A - 6, 30b + %A + 5%A + 30%A + 6, 
  30c + %A + 5%A + 30%A + 6, 30d + %A + 5%A + 30%A + 6, 
  30e + %A + 5%A + 30%A + 6, 30f + %A + 5%A + 6]
"

"complexRoots= ? - 6? + 30? - 36? + 36,
coordinates = 
[6a - %A + 5%A - 24%A + 6, 30b + %A - 5%A + 30%A - 6, 
  30c + %A - 5%A + 30%A - 6, 30d + %A - 5%A + 30%A - 6, 
  30e + %A - 5%A + 30%A - 6, 30f + %A - 5%A - 6]
"

"complexRoots= ? + 12? + 144,
coordinates = 
[12a + %A + 12, 12b + %A + 12, 12c + %A + 12, 12d + %A + 12, 
  6e - %A + 3%A - 12, 6f - %A - 3%A - 12]
--R ]
--R ,
--R 2
--R [complexRoots= \( ? - 12 \),
--R coordinates= \( [a - 1, b - 1, c - 1, d - 1, 2e + %A + 4, 2f - %A + 4] \)]
--R ,
--R 2
--R [complexRoots= \( ? + 6? + 6 \),
--R coordinates= \( [a + %A + 5, b - 1, c - 1, d - 1, e - 1, f - %A - 1] \)]
--R ,
--R 2
--R [complexRoots= \( ? - 6? + 6 \),
--R coordinates= \( [a + %A - 5, b + 1, c + 1, d + 1, e + 1, f - %A + 1] \)]
--R ,
--R 2
--R [complexRoots= \( ? - 12 \),
--R coordinates= \( [a + 1, b + 1, c + 1, d + 1, 2e + %A - 4, 2f - %A - 4] \)]
--R ,
--R 4 3 2
--R [complexRoots= \( ? + 6? + 30? + 36? + 36 \),
--R coordinates =
--R 3 2
--R [30a - %A - 5%A - 30%A - 6, 30b - %A - 5%A - 30%A - 6,
--R 3 2
--R 30c - %A - 5%A - 30%A - 6, 6d + %A + 5%A + 24%A + 6,
--R 3 2
--R 30e - %A - 5%A - 6, 30f - %A - 5%A - 30%A - 6]
--R ]
--R ,
--R 4 3 2
--R coordinates =
--R 3 2
--R [30a - %A + 5%A - 30%A + 6, 30b - %A + 5%A - 30%A + 6,
--R 3 2
--R 30c - %A + 5%A - 30%A + 6, 6d + %A - 5%A + 24%A - 6,
--R 3 2
--R 30e - %A + 5%A + 6, 30f - %A + 5%A - 30%A + 6]
--R ]
--R ,
--R 2
--R [complexRoots= ? + 6? + 6,
--R coordinates= [a + 1,b + 1,c + 1,d - %A - 5,e + %A + 1,f + 1]]
--R ,
--R
--R 2
--R [complexRoots= ? - 6? + 6,
--R coordinates= [a - 1,b - 1,c - 1,d - %A + 5,e + %A - 1,f - 1]]
--R ]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer))))
--E 21

--S 22 of 22
concat [realSolve(ts)$zdpack for ts in lts]
--R
--R
--R (22)
--R [[%B1,%B1,%B1,%B5,- %B5 - 4%B1,%B1], [%B1,%B1,%B1,%B6,- %B6 + 4%B1,%B1],
--R [%B7,%B7,%B7,%B11, - %B11 - 4%B7], [%B7,%B7,%B7,%B7,- %B12 + 4%B7],
--R [%B8,%B8,%B8,%B9, - %B9 + 4%B8], [%B8,%B8,%B8,%B8,- %B10 - 4%B8],
--R [%B13,%B13,%B17,- %B17 - 4%B13,%B13], [%B13,%B13,%B13,- %B18 - 4%B13,%B13],
--R [%B14,%B14,%B15,- %B15 - 4%B14,%B14], [%B14,%B14,%B16,- %B16 - 4%B14,%B14],
--R [%B19, %B29, 7865521 31 6696179241 25 25769893181 19
--R 6006689520 2002229840 49235160
--R +
--R 1975912990729 13 1048460696489 7 21252634831
--R - %B19 - %B19 - %B19
--R 3003344760 2002229840 6006689520
--R ,
--R
--R 778171189 31 1987468196267 25 155496778477189 19
--R %B19 + %B19 + %B19
--R 1387545279120 1387545279120 693772639560
--R +
--R 191631411158401 13 300335488637543 7 755656433863
--R %B19 + %B19 + %B19
--R 693772639560 1387545279120 198220754160
--R ,
--R
--R 1094352947 31 2794979430821 25 218708802908737 19
--R %B19 - %B19 - %B19
--R 462515093040 462515093040 231257546520
--R +
--R 91476663003591 13 145152550961823 7 1564893370717
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
--R 1387545279120 1387545279120
--R ]
--R ,
--R [%B20, %B28,
--R 7865521 31 6696179241 25 25769893181 19
--R ------------ %B20 - ------------ %B20 - ------------ %B20
--R 6006689520 2002229840 49235160
--R +
--R 1975912990729 13 1048460696489 7 21252634831
--R - ------------ %B20 - ------------ %B20 - ------------ %B20
--R 3003344760 2002229840 6006689520
--R ,
--R 778171189 31 1987468196267 25 155496778477189 19
--R - ------------ %B20 + ------------ %B20 + ------------ %B20
--R 1387545279120 1387545279120 693772639560
--R +
--R 1916314111158401 13 300335488637543 7 756656433863
--R - ------------ %B20 + ------------ %B20 - ------------ %B20
--R 693772639560 1387545279120 198220754160
--R ,
--R 1094352947 31 2794979430821 25 218708802908737 19
--R ------------ %B20 - ------------ %B20 - ------------ %B20
--R 462515093040 462515093040 231257546520
--R +
--R 91476663003591 13 145152550961823 7 1564893370717
--R - ------------ %B20 - ------------ %B20 - ------------ %B20
--R 77085848840 154171697680 462515093040
--R ,
--R 4321823003 31 180949546069 25
--R - %B28 - ------------ %B20 + ------------ %B20
--R 1387545279120 22746643920
--R +
--R 863753195062493 19 1088094465732317 13
--R - ------------ %B20 + ------------ %B20
--R 693772639560 693772639560
--R +
--R 1732620732685741 7 13506088516033
--R - ------------ %B20 + ------------ %B20
--R 1387545279120 1387545279120
--R ]
--R ,
--R [%B21, %B25,
--R 7865521 31 6696179241 25 25769893181 19
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>693772639560</td>
<td>1387545279120</td>
<td>198220754160</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1094352947</td>
<td>31</td>
<td>2794979430821</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>462515093040</td>
<td>462515093040</td>
<td>231257546520</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>91476663003591</td>
<td>13</td>
<td>145152550961823</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>77085848840</td>
<td>462515093040</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4321823003</td>
<td>31</td>
<td>180949546069</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1387545279120</td>
<td>22746643920</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>863753195062493</td>
<td>19</td>
<td>1088094456732317</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>693772639560</td>
<td>693772639560</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1975912990729</td>
<td>13</td>
<td>1048460696489</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3003344760</td>
<td>2002229840</td>
<td>6006689520</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1387545279120</td>
<td>1387545279120</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7865521</td>
<td>31</td>
<td>6696179241</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>6006689520</td>
<td>2002229840</td>
<td>49235160</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>191631411158401</td>
<td>13</td>
<td>300335488637543</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>693772639560</td>
<td>1387545279120</td>
<td>198220754160</td>
<td></td>
</tr>
<tr>
<td></td>
<td>778171189</td>
<td>31</td>
<td>1987468196267</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1387545279120</td>
<td>1387545279120</td>
<td>693772639560</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>191631411158401</td>
<td>13</td>
<td>300335488637543</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>693772639560</td>
<td>1387545279120</td>
<td>198220754160</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1094352947</td>
<td>31</td>
<td>2794979430821</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>462515093040</td>
<td>462515093040</td>
<td>231257546520</td>
<td></td>
</tr>
</tbody>
</table>
--R 91476663003591 13 145152550961823 7 1564893370717
--R - ----------- %B22 - ----------- %B22 - ----------- %B22
--R 77085848840 154171697680 462515093040
--R,
--R 4321823003 31 180949546069 25
--R - %B24 - ----------- %B22 + ----------- %B22
--R 1387545279120 22746643920
--R +
--R 863753195062493 19 1088094456732317 13
--R ----------- %B22 + ----------- %B22
--R 693772639560 693772639560
--R +
--R 1732620732685741 7 13506088516033
--R ----------- %B22 + ----------- %B22
--R 1387545279120 1387545279120
--R]
--R,
--R [%B31,%B35,- %B35 - 4%B31,%B31,%B31,%B31],
--R [%B31,%B36,- %B36 - 4%B31,%B31,%B31,%B31],
--R [%B32,%B33,- %B33 - 4%B32,%B32,%B32,%B32],
--R [%B32,%B34,- %B34 - 4%B32,%B32,%B32,%B32]
--R Type: List(List(RealClosure(Fraction(Integer))))
--E 22
)spool
)lisp (bye)

---

LexTriangularPackage.help

LexTriangularPackage examples

A package for solving polynomial systems with finitely many solutions.

The decompositions are given by means of regular triangular sets.
The computations use lexicographical Groebner bases. The main operations
are lexTriangular and squareFreeLexTriangular. The second one provide
decompositions by means of square-free regular triangular sets.

Both are based on the lexTriangular method described in
D. LAZARD "Solving Zero-dimensional Algebraic Systems"

They differ from the algorithm described in
M. MORENO MAZA and R. RIODBQ "Computations of gcd over
algebraic towers of simple extensions"
by the fact that multiplicities of the roots are not kept. With the squareFreeLexTriangular operation all multiplicities are removed. With the other operation some multiplicities may remain. Both operations admit an optional argument to produce normalized triangular sets.

The LexTriangularPackage package constructor provides an implementation of the lexTriangular algorithm (D. Lazard "Solving Zero-dimensional Algebraic Systems", J. of Symbol. Comput., 1992). This algorithm decomposes a zero-dimensional variety into zero-sets of regular triangular sets. Thus the input system must have a finite number of complex solutions. Moreover, this system needs to be a lexicographical Groebner basis.

This package takes two arguments: the coefficient-ring R of the polynomials, which must be a GcdDomain and their set of variables given by ls a List Symbol. The type of the input polynomials must be NewSparseMultivariatePolynomial(R,V) where V is OrderedVariableList(ls). The abbreviation for LexTriangularPackage is LEXTRIPK. The main operations are lexTriangular and squareFreeLexTriangular. The later provide decompositions by means of square-free regular triangular sets, built with the SquareFreeRegularTriangularSet constructor, whereas the former uses the RegularTriangularSet constructor. Note that these constructors also implement another algorithm for solving algebraic systems by means of regular triangular sets; in that case no computations of Groebner bases are needed and the input system may have any dimension (i.e. it may have an infinite number of solutions).

The implementation of the lexTriangular algorithm provided in the LexTriangularPackage constructor differs from that reported in "Computations of gcd over algebraic towers of simple extensions" by M. Moreno Maza and R. Rioboo (in proceedings of AAECC11, Paris, 1995). Indeed, the squareFreeLexTriangular operation removes all multiplicities of the solutions (i.e. the computed solutions are pairwise different) and the lexTriangular operation may keep some multiplicities; this later operation runs generally faster than the former.

The interest of the lexTriangular algorithm is due to the following experimental remark. For some examples, a triangular decomposition of a zero-dimensional variety can be computed faster via a lexicographical Groebner basis computation than by using a direct method (like that of SquareFreeRegularTriangularSet and RegularTriangularSet). This happens typically when the total degree of the system relies essentially on its smallest variable (like in the Katsura systems). When this is not the case, the direct method may give better timings (like in the Rose system).

Of course, the direct method can also be applied to a lexicographical Groebner basis. However, the lexTriangular algorithm takes advantage of the structure of this basis and avoids many unnecessary computations.
which are performed by the direct method.

For this purpose of solving algebraic systems with a finite number of solutions, see also the ZeroDimensionalSolvePackage. It allows to use both strategies (the lexTriangular algorithm and the direct method) for computing either the complex or real roots of a system.

Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

Since the LexTriangularPackage package constructor is limited to zero-dimensional systems, it provides a zeroDimensional? operation to check whether this requirement holds. There is also a groebner operation to compute the lexicographical Groebner basis of a set of polynomials with type NewSparseMultivariatePolynomial(R,V). The elimination ordering is that given by ls (the greatest variable being the first element of ls). This basis is computed by the FLG algorithm (Faugere et al. "Efficient Computation of Zero-Dimensional Groebner Bases by Change of Ordering", J. of Symbol. Comput., 1993) implemented in the LinGroebnerPackage package constructor.

Once a lexicographical Groebner basis is computed, then one can call the operations lexTriangular and squareFreeLexTriangular. Note that these operations admit an optional argument to produce normalized triangular sets. There is also a zeroSetSplit operation which does all the job from the input system; an error is produced if this system is not zero-dimensional.

Let us illustrate the facilities of the LexTriangularPackage constructor by a famous example, the cyclic-6 root system.

Define the coefficient ring.

\begin{verbatim}
R := Integer
   Type: Domain
\end{verbatim}

Define the list of variables,

\begin{verbatim}
ls : List Symbol := [a,b,c,d,e,f]
   Type: List Symbol
\end{verbatim}

and make it an ordered set.

\begin{verbatim}
V := OVAR(ls)
   OrderedVariableList [a,b,c,d,e,f]
   Type: Domain
\end{verbatim}

Define the polynomial ring.
\[ P := \text{NSMP}(R, V) \]
\[ \text{NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]
\[ \text{Type: Domain} \]

Define the polynomials.

- \[ p_1: P := a*b*c*d*e*f - 1 \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

- \[ p_2: P := a*b*c*d*e + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f \]
  \[ (((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

- \[ p_3: P := a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f \]
  \[ (c + f)b + f e)a + d c b + e d c + f e d \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

- \[ p_4: P := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f \]
  \[ (b + f)a + c b + d c + e d + f e \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

- \[ p_5: P := a*b + a*f + b*c + c*d + d*e + e*f \]
  \[ a + b + c + d + e + f \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

- \[ p_6: P := a*b + c*d + e*f \]
  \[ a + b + c + d + e + f \]
  \[ \text{Type: NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

\[ l p := [p_1, p_2, p_3, p_4, p_5, p_6] \]
\[ [f e d c b a - 1, ((((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b, ((d + f)c + f e)b + f e d c)a + e d c b + f e d c, (c + f)b + f e)a + d c b + e d c + f e d, (b + f)a + c b + d c + e d + f e, a + b + c + d + e + f] \]
\[ \text{Type: List NewSparseMultivariatePolynomial}(\text{Integer}, \text{OrderedVariableList} \ [a,b,c,d,e,f]) \]

Now call LEXTRIPK.

\[ \text{lextripack} := \text{LEXTRIPK}(R, l s) \]
\[ \text{LexTriangularPackage}(\text{Integer}, [a,b,c,d,e,f]) \]
\[ \text{Type: Domain} \]
Compute the lexicographical Groebner basis of the system. This may take between 5 minutes and one hour, depending on your machine.

\[
lg := \text{groebner(lp)}; \text{lextripack}\\
[a + b + c + d + e + f,\\
\begin{align*}
2 & \quad 3968379498283200b + 15873517993132800f b + 3968379498283200d \\
+ & \quad 3 \quad 5 \\
\begin{align*}
15873517993132800f d & + 3968379498283200f e - 15873517993132800f e \\
+ & \quad 5 \quad 3 \\
\begin{align*}
23810276989699200f e & + (206355733910726400f + 230166010900425600)e \\
+ & \quad 43 \quad 37 \\
- 729705987316687f & + 1863667496867205421f \\
+ & \quad 31 \\
291674853771731104461f & + 365285994691106921745f \\
+ & \quad 19 \quad 13 \\
549961185828911895f & - 365048404038768439269f \\
+ & \quad 7 \\
- 292382820431504027669f & - 2271898467631865497f \\
\end{align*}
\end{align*}
\end{align*}
\]
\[(- 2428648252949318400f - 2611193709870345600)e + 8305444561289527f - 21212087151945459641f + 331981583093451385381f - 4157691646261667136445f + 6072721607510764095f + 4154986709036460221649f + 3327761311138587096749f + 25885340608290841637f \times e + 45815897629010329f - 1170137655582161891207f + 18313166848970865074187f - 22909971239649297438915f + 16133250761305157265f + 22897305857636178256623f + 18329944781867242497923f + 130258531002020420699f, (7936758996566400d - 7936758996566400f)b - 7936758996566400f d + 3\times 5\times 4\times 4\times 5\times 3 - 7936758996566400f e + 23810276989699200f e - 23810276989699200f e + 6\times 2\times (- 337312257364072000f - 369059293340337600)e + 1176345388640471f - 3004383582891473073f + 470203502707246105653f - 588858183402644348085f + 856939308623513535f + 588472674242340526377f + 471313241958371103517f + 3659742549078552381f \times e}
\[
\begin{align*}
&\quad + 44 \quad 38 \quad 32 \\
&\quad 6423170513956901f - 16404772137036480803f - 25674191652275287774463f \\
&\quad + 26 \quad 20 \\
&\quad - 3211938090825682172335f - 2330490332697587485f \\
&\quad + 14 \quad 8 \\
&\quad 3210100109444754864587f + 2569858315395162617847f \\
&\quad + 2 \\
&\quad 18326089487427735751f \\
&\quad , \\
&\quad (11905138494849600e - 11905138494849600f)b - 3968379498283200f e \\
&\quad + 44 \quad 38 \quad 32 \\
&\quad 4177638546747827f - 10669685294602576381f - 1669852980419949524601f \\
&\quad + 26 \quad 20 \\
&\quad - 2089077057287904170745f - 1659899763580278795f \\
&\quad + 14 \quad 8 \\
&\quad 2087864026859015573349f + 1671496085945199577969f \\
&\quad + 2 \\
&\quad 11940257226216280177f \\
&\quad , \\
&\quad (11905138494849600f - 11905138494849600)b - 15873517993132800f e 
\end{align*}
\]
\[
\begin{align*}
& 39683794982832000f e - 39683794982832000f e \\
& + \quad 11 \quad 5 \quad 2 \\
& - 686529653202993600f e - 607162063237329600f e \\
& + \quad 42 \quad 36 \quad 30 \\
& 65144531306704f e - 166381280901088652f e - 26033434502470283472f e \\
& + \quad 24 \quad 18 \\
& - 31696259583860650140f + 971492093167581360f \\
& + \quad 12 \quad 6 \\
& 322008503369189548f + 25526177666070529808f + 138603268355749244 \\
& \times e \\
& + \quad 43 \quad 37 \quad 31 \\
& 167620036074811f - 428102417974791473f - 66997243801231679313f \\
& + \quad 25 \quad 19 \\
& - 83426716722148750485f + 203673895369980765f \\
& + \quad 13 \quad 7 \\
& 83523056326010432457f + 66995789640238066937f + 47859285549587901f \\
& + \quad 3 \quad 2 \quad 2 \\
& 801692827936c + 2405078483808f c - 2405078483808f c - 13752945467f \\
& + \quad 39 \quad 33 \quad 27 \\
& 35125117815561f + 5496946957826433f + 6834659447749117f \\
& + \quad 21 \quad 15 \quad 9 \\
& - 44484880462461f - 6873406230093057f - 5450844938762633f \\
& + \quad 3 \\
& 1216586044571f \\
& + \quad 2 \\
& (23810276989699200d - 23810276989699200d) c + 23810276989699200d \\
& + \quad 3 \quad 5 \quad 4 \\
& 71430830969097600f d + 7936758996566400f e - 31747035986265600f e \\
& + \quad 5 \quad 3 \quad 6 \\
& 31747035986266600f e + (404774708824886400f + 396837949828320000f) e \\
& + \quad 43 \quad 37 \\
& - 1247372229446701f + 3185785654596621203f
\end{align*}
\]
\[
\begin{align*}
&+ \quad 31 \quad 25 \\
&\quad 49859486684974751463f + 624542545845791047935f \\
&\quad + \quad 19 \quad 13 \\
&\quad 931085755769682885f - 624150663582417063387f \\
&\quad + \quad 7 \\
&\quad - 49988185938360475647f - 3926885313819527351f \\
&\times e \\
&+ \quad 44 \quad 38 \\
&\quad - 7026011547118141f + 17944427051950691243f \\
&\quad + \quad 32 \quad 26 \\
&\quad 2808383522593986603543f + 3513624142354807530135f \\
&\quad + \quad 20 \quad 14 \\
&\quad 286075006705537685f - 3511356735642190737267f \\
&\quad + \quad 8 \quad 2 \\
&\quad - 2811332494697103819887f - 20315011631522847311f \\
&\quad \times e \\
&+ \quad 44 \quad 38 \quad 31 \\
&\quad - 4418748183673f + 1128556870456559f + 1765998617294451019f \\
&\quad + \quad 25 \quad 19 \\
&\quad 2173749283622606155f - 55788292195402895f \\
&\quad + \quad 13 \quad 7 \\
&\quad - 2215291421788292951f - 1718142665347430851f + 3025669458230237f \\
&\times e \\
&+ \quad 44 \quad 38 \quad 32 \\
&\quad 4418748183673f - 11285568707456559f - 1765998617294451019f \\
&\quad + \quad 26 \quad 20 \quad 14 \\
&\quad - 2173749283622606155f + 55788292195402895f + 2215291421788292951f \\
&\quad + \quad 8 \quad 2 \\
&\quad 1718142665347430851f - 3025669458230237f \\
&\quad \times (72152354514240f - 72152354514240)c + 40950859449f \\
&+ \quad 37 \quad 31 \quad 25 \\
\end{align*}
\]
\[
\begin{align*}
& - 104588980990367f - 16367227395575307f - 20268523416527355f \\
& + 19 13 7 \\
& 442205002259535f + 20576059935789063f + 1599713796970563f \\
& + - 27509982785581f \\
& , 3 2 2 \\
& 1984189749141600d + 5952569247424800f d - 5952569247424800f d \\
& + 4 5 5 4 3 \\
& - 3968379498283200f e + 15873517993132800f e + 17857707742274400e \\
& + 7 2 \\
& (- 148814231185620000f - 16270355942961200f) e \\
& + 44 38 \\
& - 390000914678878f + 996062704593756434f \\
& + 32 26 \\
& 15586323972034823914f + 194745956143985421330f \\
& + 20 14 \\
& 6205077595574430f - 19459651265329906878f \\
& + 8 2 \\
& - 155796897940756922666f - 1036375759077320978f \\
* e \\
& + 45 33 \\
& - 374998630035991f + 957747106595653993f + 149889155566764891693f \\
& + 27 21 \\
& 187154171443494641685f - 127129015426348065f \\
& + 15 9 \\
& - 18724153243115040417f - 149719985667976534037f - 836654081239648061f \\
, 3 5 \\
& (5952569247424800e - 5952569247424800f)d - 3968379498283200f e \\
& + 44 4 5 3 \\
& 9920948745708000f e - 3968379498283200f e \\
& + 6 2 \\
& (- 148814231185620000f - 150798420934761600f)e \\
& + 43 37 \\
& 492558110242553f - 1257992359608074599f
\end{align*}
\]
PACKAGE LEXTRIPK LEXTRIANGULARPACKAGE

+ 31 25
- 19688309453968513959f - 246562115745735428055f
+ 19 13
- 3256987019938885505f + 246417769883661808111f
+ 7
19732735208200652911f + 152337379638932143f
* e
+ 44 38 32
2679481081803026f - 6843392695421906608f - 1071020459642646913578f
+ 26 20
- 1339789169692041240060f - 852746750910750210f
+ 14 8
1339105101971878401312f + 1071900289758712987462f
+ 2
7555239072072727756f
, 6 2 5
(11905138494849600f - 11905138494849600d - 793675899666400f e
+ 3 4 4 3
31747035986265600f e - 31747035986265600f e
+ 11 5 2
(- 420648226818019200f - 404774708824886400f )e
+ 42 36 30
153361876000889f - 39169739565161107f - 6127176127489690827f
+ 24 18
- 7217708742310509615f + 538628483890722735f
+ 12 6
7506804353843507643f + 5886160769782607203f + 63576108396535879
* e
+ 43 37 31
71737781777066f - 183218856207557938f - 28672874271132276078f
+ 25 19
- 3562522366939812010f + 164831339634084390f
+
CHAPTER 13. CHAPTER L

\[ \begin{align*}
&13 \cdot \frac{7}{12} \cdot \frac{5}{6} \cdot \frac{24}{23} \cdot \frac{3}{3} \cdot \frac{4}{3} \cdot \frac{2}{10} \cdot \left( -\frac{230166010900425600}{e} + \frac{226197631402142400}{e} \right) \\
&+ \frac{47}{46} \cdot \frac{29}{35} \cdot \frac{17}{23} \cdot \frac{5}{11} \cdot \frac{7}{6} \cdot \frac{2}{42} \cdot \frac{30}{36} \cdot \frac{18}{24} \cdot \frac{27}{12} \cdot \frac{21}{6} \cdot \frac{3}{25438330117200} + \frac{3}{25438330117200} \\
&+ \frac{44}{43} \cdot \frac{32}{38} \cdot \frac{20}{26} \cdot \frac{14}{8} \cdot \frac{2}{2} \cdot \left( -\frac{638132320196044965}{e} + \frac{4510507167940725}{e} \right) \\
&+ \frac{35724160423073052642}{e} + \frac{28627022578664910622}{e} + \frac{187459987029680506}{e} \\
&+ \frac{1322793166094400}{e} - \frac{3968379498283200}{e} + \frac{3968379498283200}{e} \\
&+ \frac{5291172664377600}{e} \\
&+ \frac{(-230166010900425600)}{e} - \frac{226197631402142400}{e} \\
&+ \frac{152375364610443885}{e} + \frac{389166626064854890415}{e} \\
&+ \frac{60906097841360558987335}{e} + \frac{76167367934608798697275}{e} \\
&+ \frac{27855066785995181125}{e} - \frac{76144952817052723145495}{e} \\
&+ \frac{60933629892463517546975}{e} - \frac{411415071682002547795}{e} \\
&\times e \\
&+ \frac{209493533143822}{e} + \frac{535045979490560586}{e} + \frac{83737947964973553146}{e} \\
&+ \frac{104889507084213371570}{e} + \frac{167117997269207870}{e} \\
&+ \frac{1047937257813990615514}{e} - \frac{83842685189903180394}{e} - \frac{569978796672974242}{e} \\
&+ \frac{(25438330117200)}{e} + \frac{25438330117200}{e} \\
&+ \frac{(76314990351600)}{e} + \frac{76314990351600}{e} \\
&+ \frac{1594966552735}{e} + \frac{4073543370415745}{e} + \frac{637527159231148925}{e} \\
&+ \frac{797521176113606525}{e} + \frac{530440941097175}{e} - \frac{797160527306433145}{e} \\
&+ \frac{638132320196044965}{e} - \frac{4510507167940725}{e} \\
&\times e
\end{align*} \]
Type: List NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])

Apply lexTriangular to compute a decomposition into regular triangular sets. This should not take more than 5 seconds.

```
lexTriangular(lg,false)$lextripack
[6 6 5 2 4 3 3 4 2 5
 {f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,
  2 5 3 4 4 3 5 2
 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
  2 5 3 4 4 3 5 2
 3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f,
]```
\[ a - f e + 3 f e - 3 f e + 4 f e + 3 e - 3 f \] 
\[ f - 1, e - f, d - f, c + 4 f, c + f, (c - f) b - f c - 5 f, a + b + c + 3 f \] 
\[ f - 1, e - f, d - f, c - f, b + 4 f, b + f, a + b + 4 f \] 
\[ f - 1, e - f, d + 4 f, d + f, (d - f) c - f d - 5 f, b - f, a + c + d + 3 f \] 
\[ f - 2554 f - 399709 f - 502276 f - 399709 f - 2554 f + 1, \] 
\[ (16171856 f - 16171856) e + 31 \] 
\[ - 504205 f + 1287737951 f + 201539391380 f + 253982817368 f + 7 \] 
\[ 201940704665 f + 157413460 f \] 
\[ * e \] 
\[ + 32 \] 
\[ - 2818405 f + 7198203911 f + 1126548149060 f + 1416530636364 f + 8 \] 
\[ 1127377589346 f + 7988820725 f \] 
\[ , 6 \] 
\[ (693772639560 f - 693772639560) d - 462515093040 f e + 1850060372160 f e + 43 \] 
\[ 3 \] 
\[ - 1850060372160 f e + (- 24513299931120 f - 23588269745040 f )e + 30 \] 
\[ 24 \] 
\[ - 890810428 f + 2275181044754 f + 355937263869776 f + 12 \] 
\[ 6 \] 
\[ 413736880104344 f + 342849304487996 f + 3704966481878 f * e + 31 \] 
\[ - 4163798003 f + 10634395752169 f + 1664161760192806 f + 13 \] 
\[ 7 \] 
\[ 2079424391370694 f + 1668153650635921 f + 10924274392693 f \] 
\[ , 6 \] 
\[ (12614047992 f - 12614047992) c - 7246825 f + 18508536599 f \]
Note that the first set of the decomposition is normalized (all initials are integer numbers) but not the second one (normalized triangular sets are defined in the description of the NormalizedTriangularSetCategory constructor).

So apply now lexTriangular to produce normalized triangular sets.

```plaintext
lts := lexTriangular(lg,true)$lextripack

[6 6 5 2 4 3 3 4 2 5
{f + 1, e - 3 f e + 3 f e - 4 f e + 3 f e - 3 f e - 1,
  2 5 3 4 4 3 5 2
3 d + f e - 4 f e + 4 f e - 2 f e - 2 e + 2 f, c + f,
  2 5 3 4 4 3 5 2
3 b + 2 f e - 5 f e + 5 f e - 10 f e - 4 e + 7 f,
  2 5 3 4 4 3 5 2
a - f e + 3 f e - 3 f e + 4 f e + 3 e - 3 f}

6 2 2
{f - 1, e - f, d - f, c + 4 f c + f, b + c + 4 f, a - f},
```
\{f - 1, e - f, d - f, c - f, b + 4f, b + f, a + b + 4f\},
\{f - 1, e - f, d + 4f, d + f, c + d + 4f, b - f, a - f\},
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, 2\}
1387545279120e
\{31, 25, 19\}
4321823003f - 11037922310209f - 1727506390124986f +
\{13, 7\}
- 2176188913464634f - 173262073265741f - 13560888516033f* e
\{32, 26, 20\}
24177661775f - 61749727185325f - 9664082618092450f +
\{14, 8, 2\}
- 12152237485813570f - 9672870290826025f - 68544102808525f,
1387545279120d
\{30, 24, 18\}
- 1128983050f + 2883434331830f + 451234998755840f +
\{12, 6\}
562426491685760f + 47129055314890f - 165557857270* e
\{31, 25, 19\}
- 1816935351f + 4640452214013f + 726247129626942f +
\{13, 7\}
912871801716798f + 726583262666877f + 4909358645961f,
1387545279120c + 778171189f - 1987468196267f - 310993556954378f +
\{13, 7\}
- 383262222316802f - 300335488637543f + 5289595037041f,
1387545279120b
+ 1128983050f - 2883434331830f - 451234998755840f +
We check that all initials are constant.

\[
\text{squareFreeLexTriangular}(\text{lg}, \text{true})$\text{lestripack}$
\]
\[ a - f e + 3f e - 3f e + 4f e + 3e - 3f \}
\[ 6 \quad 2 \quad 2 \]
\[ \{f - 1, e - f, d - f, c + 4f, c + f, b + c + 4f, a - f \}, \]
\[ 6 \quad 2 \quad 2 \]
\[ \{f - 1, e - f, d - f, c - f, b + 4f, b + f, a + b + 4f \}, \]
\[ 6 \quad 2 \quad 2 \]
\[ \{f - 1, e - f, d + 4f, d + f, c + d + 4f, b - f, a - f \}, \]
\[ 36 \quad 30 \quad 24 \quad 18 \quad 12 \quad 6 \]
\[ \{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \]
\[ 2 \quad 13 \quad 8 \quad 2 \]
\[ 1387545279120e \]
\[ + \]
\[ 31 \quad 25 \quad 19 \]
\[ 4321823003f - 11037922310209f - 1727506390124986f \]
\[ + \]
\[ 13 \quad 7 \]
\[ 2176188913464634f - 1732620732685741f - 13506088516033f \]
\[ * \]
\[ 1387545279120d \]
\[ + \]
\[ 32 \quad 26 \quad 20 \]
\[ 24177661775f - 61749727185325f - 9664082618902450f \]
\[ + \]
\[ 14 \quad 8 \quad 2 \]
\[ - 12152237485813570f - 9672870290826025f - 68544102808525f \]
\[ , \]
\[ 1387545279120e \]
\[ + \]
\[ 30 \quad 24 \quad 18 \]
\[ - 1128983050f + 2883434331830f + 451234998755840f \]
\[ + \]
\[ 12 \quad 6 \]
\[ 562426491685760f + 447129055314890f - 165557857270 \]
\[ * \]
\[ 1387545279120f \]
\[ + \]
\[ 31 \quad 25 \quad 19 \]
\[ - 1816935351f + 4640452214013f + 726247129626942f \]
\[ + \]
\[ 13 \quad 7 \]
\[ 912871801716798f + 726583262666877f + 4909358645961f \]
\[ , \]
\[ 1387545279120g \]
\[ + \]
\[ 31 \quad 25 \quad 19 \]
\[ - 1897468196267f - 310993556954378f \]
\[ + \]
\[ 13 \quad 7 \]
\[ - 383262822316802f - 30035488637543f + 5289595037041f \]
\[ , \]
\[ 1387545279120h \]
Thus the solutions given by lts are pairwise different.

We count them as follows.

reduce(+,[degree(ts) for ts in lts])

156

Type: PositiveInteger

We can investigate the triangular decomposition lts by using the
ZeroDimensionalSolvePackage.

This requires to add an extra variable (smaller than the others) as follows.

ls2 : List Symbol := concat(ls,new()$Symbol)

[a,b,c,d,e,f,%A]

Type: List Symbol

Then we call the package.
zdpack := ZDSOLVE(R,ls,ls2)
(20) ZeroDimensionalSolvePackage(Integer,[a,b,c,d,e,f],[a,b,c,d,e,f,%A])
Type: Domain

We compute a univariate representation of the variety associated with
the input system as follows.

concat [univariateSolve(ts)$zdpack for ts in lts]
[
4 2
[complexRoots= ? - 13? + 49,
coordinates =
3 3 3 3
[7a + %A - 6%A, 21b + %A + %A, 21c - 2%A + 19%A, 7d - %A + 6%A,
3 3
21e - %A - %A, 21f + 2%A - 19%A]
]
,
4 2
[complexRoots= ? + 11? + 49,
coordinates =
3 3 3
[35a + 3%A + 19%A, 35b + %A + 18%A, 35c - 2%A - %A,
3 3 3
35d - 3%A - 19%A, 35e - %A - 18%A, 35f + 2%A + %A]
]
,
[
complexRoots =
8 7 6 5 4 3 2
,coordinates =
7 6 5 4
43054532a + 33782%A - 546673%A + 3127348%A - 6927123%A
+ 3 2
4365212%A - 25086957%A + 39582814%A - 107313172
, 7 6 5 4
43054532b - 33782%A + 546673%A - 3127348%A + 6927123%A
+ 3 2
- 4365212%A + 25086957%A - 39582814%A + 107313172
, 7 6 5 4
21527266c - 22306%A + 263139%A - 1166076%A + 1821805%A
+
The image contains a mathematical expression and some text. The mathematical expression seems to be a series of complex numbers or polynomials. The text on the image is as follows:

```

complexRoots = [3 2]
  a + b^2 - c^2 + d^2 + e^2 + f^2 + g^2 + h^2 + i^2 + j^2 + k^2 + l^2 + m^2 + n^2 + o^2 + p^2 + q^2 + r^2 + s^2 + t^2 + u^2 + v^2 + w^2 + x^2 + y^2 + z^2

coordinates = [7 6 5 4 3 2]

a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v + w + x + y + z
```

The text appears to be discussing complex roots and coordinates, possibly in a mathematical or computational context.
+ 3 2
2892788%A + 10322663%A + 30553862%A + 12950740
, 7 6 5 4
43054532e - 22306%A - 263139%A - 1166076%A - 1821805%A
+ 3 2
- 2892788%A - 10322663%A - 30553862%A - 12950740
, 7 6 5 4
21527266f + 22306%A + 263139%A + 1166076%A + 1821805%A
+ 3 2
2892788%A + 10322663%A + 9026596%A + 12950740
]
]

[complexRoots= ? - ? + 1,
3 3 3 3
coordinates= [a - %A,b + %A - %A,c + %A ,d + %A,e - %A + %A,f - %A ]
]
, 8 6 4 2
[complexRoots= ? + 4? + 12? + 16? + 4,
coordinates =
7 5 3 7 5 3
[4a - 2%A - 7%A - 20%A - 22%A, 4b + 2%A + 7%A + 20%A + 22%A,
7 5 3 7 5 3
4c + %A + 3%A + 10%A + 10%A, 4d + %A + 3%A + 10%A + 6%A,
7 5 3 7 5 3
4e - %A - 3%A - 10%A - 6%A, 4f - %A - 3%A - 10%A - 10%A]
]
, 4 3 2
[complexRoots= ? + 6? + 30? + 36? + 36,
coordinates =
3 2
[30a - %A - 5%A - 30%A - 6, 6b + %A + 5%A + 24%A + 6,
3 2
30c - %A - 5%A - 6, 30d - %A - 5%A - 30%A - 6,
3 2
30e - %A - 5%A - 30%A - 6, 30f - %A - 5%A - 30%A - 6]
]
, 4 3 2
[complexRoots= ? - 6? + 30? - 36? + 36,
coordinates =
3 2
[30a - %A + 5%A - 30%A + 6, 6b + %A - 5%A + 24%A - 6,
\[
\begin{align*}
3 & 2 \\
30c - \%A + 5\%A + 6, & 30d - \%A + 5\%A - 30\%A + 6, \\
3 & 2 \\
30e - \%A + 5\%A - 30\%A + 6, & 30f - \%A + 5\%A - 30\%A + 6.
\end{align*}
\]

\[
\begin{align*}
2 \\
\text{[complexRoots= } \pm 6? + 6, \\
\text{coordinates= } [a + 1, b - \%A - 5, c + \%A + 1, d + 1, e + 1, f + 1].
\end{align*}
\]

\[
\begin{align*}
2 \\
\text{[complexRoots= } \pm 6? + 6, \\
\text{coordinates= } [a - 1, b - \%A + 5, c - \%A - 1, d - 1, e - 1, f - 1].
\end{align*}
\]

\[
\begin{align*}
4 & 3 \\
30a - \%A - 5\%A - 30\%A - 6, & 30b - \%A - 5\%A + 30\%A - 6, \\
3 & 2 \\
30c - \%A - 5\%A - 30\%A + 6, & 30d - \%A + 5\%A + 6, \\
3 & 2 \\
\end{align*}
\]

\[
\begin{align*}
\text{[complexRoots= } \pm 6? + 30? + 36? + 36, \\
\text{coordinates= } \pm 6? + 6, \\
\text{coordinates= } [a - \%A - 1, b + \%A + 1, c + 1, d + 1, e + 1, f + 1].
\end{align*}
\]

\[
\begin{align*}
\text{[complexRoots= } \pm 6? + 6, \\
\text{coordinates= } [a - \%A + 1, b + \%A + 1, c + 1, d + 1, e + 1, f + 1].
\end{align*}
\]

\[
\begin{align*}
4 & 3 \\
\text{[complexRoots= } \pm 6? + 30? + 36? + 36, \\
\text{coordinates= } \pm 6? + 6, \\
\text{coordinates= } [a - \%A - 1, b + \%A + 1, c - 1, d - 1, e - 1, f - 1].
\end{align*}
\]
6c + %A + 5%A + 24%A + 6, 30d - %A - 5%A - 6,
3 3 2
30e - %A - 5%A - 30%A - 6, 30f - %A - 5%A - 30%A - 6]

, 4 3 2

[complexRoots= ? - 6? + 30? - 36? + 36,
coordinates =
3 2
[30a - %A + 5%A - 30%A + 6, 30b - %A + 5%A - 30%A + 6,
3 3 2
6c + %A - 5%A + 24%A - 6, 30d - %A + 5%A + 6,
3 3 2
30e - %A + 5%A - 30%A + 6, 30f - %A + 5%A - 30%A + 6]

, 2

[complexRoots= ? + 6? + 6,
coordinates= [a + 1,b + 1,c - %A - 5,d + %A + 1,e + 1,f + 1]]

, 2

[complexRoots= ? - 6? + 6,
coordinates= [a - 1,b - 1,c - %A + 5,d + %A - 1,e - 1,f - 1]]

, 8 7 6 5 4 2

coordinates =
7 6 5 4 3 2
[2a + 2%A + 9%A + 18%A + 19%A + 4%A - 10%A - 2%A + 4,
7 7 6 5 4 3 2
2b + 2%A + 9%A + 18%A + 19%A + 4%A - 10%A - 4%A + 4,
7 7 6 5 4 3
2c - %A - 4%A - 8%A - 9%A - 4%A - 2%A - 4,
7 7 6 5 4 3
2d + %A + 4%A + 8%A + 9%A + 4%A + 2%A + 4,
7 7 6 5 4 3 2
2e - 2%A - 9%A - 18%A - 19%A - 4%A + 10%A + 4%A - 4,
7 7 6 5 4 3 2
2f - 2%A - 9%A - 18%A - 19%A + 10%A + 2%A - 4]

, [
complexRoots = 8 7 6 5 4 3 2
, coordinates =
7 6 5 4 3 2
1408a - 19%A - 200%A - 912%A - 2216%A - 4544%A - 6784%A
+ 6976%A - 1792
\begin{equation}
\begin{aligned}
&1408b - 37\%A - 408\%A - 1952\%A - 5024\%A - 10368\%A - 16768\%A \\
+ &17920\%A - 5120
&1408c + 37\%A + 408\%A + 1952\%A + 5024\%A + 10368\%A + 16768\%A \\
+ &17920\%A + 5120
&1408d + 19\%A + 200\%A + 912\%A + 2216\%A + 4544\%A + 6784\%A \\
+ &6976\%A + 1792
\end{aligned}
\end{equation}

\begin{align*}
&2e + \%A, 2f - \%A
\end{align*}

\begin{align*}
&\text{complexRoots} = \begin{cases}
8 & 6 & 4 & 2 \\
\text{coordinates} = 7 & 5 & 3 & 7 & 5 & 3 \\
4a - \%A - 3\%A - 10\%A - 6\%A, 4b - \%A - 3\%A - 10\%A - 10\%A, \\
7 & 5 & 3 & 7 & 5 & 3 \\
4c - 2\%A - 7\%A - 20\%A - 22\%A, 4d + 2\%A + 7\%A + 20\%A + 22\%A, \\
7 & 5 & 3 & 7 & 5 & 3 \\
4e + \%A + 3\%A + 10\%A + 10\%A, 4f + \%A + 3\%A + 10\%A + 6\%A
\end{cases}
\end{align*}

\begin{align*}
&\text{complexRoots} = \begin{cases}
8 & 6 & 4 & 2 \\
\text{coordinates} = 7 & 5 & 3 & 7 & 5 & 3 \\
512a - \%A - 12\%A + 176\%A - 448\%A, \\
7 & 5 & 3 \\
128b - \%A - 16\%A + 96\%A - 256\%A, \\
7 & 5 & 3 \\
128c + \%A + 16\%A - 96\%A + 256\%A, \\
7 & 5 & 3 \\
512d + \%A + 12\%A - 176\%A + 448\%A, 2e + \%A, 2f - \%A
\end{cases}
\end{align*}

\begin{align*}
&\text{complexRoots} = \\
&\begin{cases}
8 & 7 & 6 & 5 & 4 & 3 & 2 \\
\end{cases}
\end{align*}
coordinates =
[
    7  6  5  4  3  2
    1408a - 19%A + 200%A - 912%A + 2216%A - 4544%A + 6784%A
    + - 6976%A + 1792
 ,
    7  6  5  4  3  2
    1408b - 37%A + 408%A - 1952%A + 5024%A - 10368%A + 16768%A
    + - 17920%A + 5120
 ,
    7  6  5  4  3  2
    1408c + 37%A - 408%A + 1952%A - 5024%A + 10368%A - 16768%A
    + 17920%A - 5120
 ,
    7  6  5  4  3  2
    1408d + 19%A - 200%A + 912%A - 2216%A + 4544%A - 6784%A
    + 6976%A - 1792
 ,
    2e + %A, 2f - %A]
]

[complexRoots= % + 12% + 144,
coordinates =
    2  2  2  2
    12a - %A - 12, 12b - %A - 12, 12c - %A - 12, 12d - %A - 12,
    2  2
    6e + %A + 3%A + 12, 6f + %A - 3%A + 12]
[[complexRoots= ? + 6? + 30? + 36? + 36, 
coordinates = 
3 3 2 3 2 
[6a - %A - 5%A - 24%A - 6, 30b + %A + 5%A + 30%A + 6, 3 3 2 3 2 
30c + %A + 5%A + 30%A + 6, 30d + %A + 5%A + 30%A + 6, 3 3 2 3 2 
30e + %A + 5%A + 30%A + 6, 30f + %A + 5%A + 6] 
] 
, 
4 3 2 
[[complexRoots= ? - 6? + 30? - 36? + 36, 
coordinates = 
3 3 2 3 2 
[6a - %A + 5%A - 24%A + 6, 30b + %A - 5%A + 30%A - 6, 3 3 2 3 2 
30c + %A - 5%A + 30%A - 6, 30d + %A - 5%A + 30%A - 6, 3 3 2 3 2 
30e + %A - 5%A + 30%A - 6, 30f + %A - 5%A - 6] 
] 
, 
4 2 
[[complexRoots= ? + 12? + 144, 
coordinates = 
2 2 2 2 2 2 
[12a + %A + 12, 12b + %A + 12, 12c + %A + 12, 12d + %A + 12, 2 2 2 2 2 2 
6e - %A + 3%A - 12, 6f - %A - 3%A - 12] 
] 
, 
2 
[[complexRoots= ? - 12, 
coordinates= [a - 1,b - 1,c - 1,d - 1,2e + %A + 4,2f - %A + 4]] 
, 
2 
[[complexRoots= ? + 6? + 6, 
coordinates= [a + %A + 5,b - 1,c - 1,d - 1,e - 1,f + %A + 1]] 
, 
2 
[[complexRoots= ? - 6? + 6, 
coordinates= [a + %A - 5,b + 1,c + 1,d + 1,e + 1,f - %A + 1]] 
, 
2 
[[complexRoots= ? - 12, 
coordinates= [a + 1,b + 1,c + 1,d + 1,2e + %A - 4,2f - %A - 4]] 
, 
4 3 2 
[[complexRoots= ? + 6? + 30? + 36? + 36,
coordinates =
 3 2
[30a - %A - 5%A - 30%A - 6, 30b - %A - 5%A - 30%A - 6,
 3 2
30c - %A - 5%A - 30%A - 6, 6d + %A + 5%A + 24%A + 6,
 3 2
30e - %A - 5%A - 6, 30f - %A - 5%A - 30%A - 6]
]

, 4 3 2
[complexRoots = ? - 6? + 30? - 36? + 36,
coordinates =
 3 2
[30a - %A + 5%A - 30%A + 6, 30b - %A + 5%A - 30%A + 6,
 3 2
30c - %A + 5%A - 30%A + 6, 6d + %A - 5%A + 24%A - 6,
 3 2
30e - %A + 5%A + 6, 30f - %A + 5%A - 30%A + 6]
]

, 2
[complexRoots = ? + 6? + 6,
coordinates = [a+1,b+1,c+1,d-%A-5,e+%A+1,f+1]]

, 2
[complexRoots = ? - 6? + 6,
coordinates = [a-1,b-1,c-1,d-%A+5,e+%A-1,f-1]]

Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
coordinates: List Polynomial Integer)

Since the univariateSolve operation may split a regular set, it
returns a list. This explains the use of concat.

Look at the last item of the result. It consists of two parts. For
any complex root ? of the univariate polynomial in the first part, we
get a tuple of univariate polynomials (in a, ..., f respectively) by
replacing %A by ? in the second part. Each of these tuples t
describes a point of the variety associated with lp by equaling to
zero the polynomials in t.

Note that the way of reading these univariate representations is
explained also in the example illustrating the ZeroDimensionalSolvePackage
constructor.

Now, we compute the points of the variety with real coordinates.

concat [realSolve(ts)$zdpack for ts in lts]
[[%B1,%B1,%B1,%B5,- %B5 - 4%B1,%B1], [%B1,%B1,%B1,%B6,- %B6 - 4%B1,%B1],
[\%B7,\%B7,\%B7,\%B7,\%B7, - \%B11 - 4\%B7], [\%B7,\%B7,\%B7,\%B7,\%B12, - \%B12 - 4\%B7],
[\%B8,\%B8,\%B8,\%B8,\%B9, - \%B9 - 4\%B8], [\%B8,\%B8,\%B8,\%B8,\%B10, - \%B10 - 4\%B8],
[\%B13,\%B13,\%B17, - \%B17 - 4\%B13,\%B13,\%B13],
[\%B13,\%B13,\%B18, - \%B18 - 4\%B13,\%B13,\%B13],
[\%B14,\%B14,\%B15, - \%B15 - 4\%B14,\%B14,\%B14],
[\%B14,\%B14,\%B16, - \%B16 - 4\%B14,\%B14,\%B14],
[\%B19, \%B29,
7865521 31 6696179241 25 25769893181 19
---------- \%B19 - ---------- \%B19 - ---------- \%B19
6006689520 2002229840 49235160
+
1975912990729 13 1048460696489 7 21252634831
- \%B19 - \%B19 - \%B19
3003344760 2002229840 6006689520

, 778171189 31 1987468196267 25 155496778477189 19
- \%B19 + \%B19 + \%B19
1387545279120 1387545279120 693772639560
+
191631411158401 13 300335488637543 7 755656433863
\%B19 + \%B19 - \%B19 - \%B19
693772639560 1387545279120 198220754160

, 1094352947 31 2794979430821 25 218708802908737 19
\%B19 - \%B19 - \%B19
462515093040 462515093040 231257546520
+
91476663003591 13 145152550961823 7 1564893370717
\%B19 - \%B19 - \%B19
77085848840 154171697680 462515093040

, 4321823003 31 180949546069 25
\%B29 - \%B19 + \%B19
1387545279120 22746643920
+
863753195062493 19 1088094456732317 13
\%B19 + \%B19
693772639560 693772639560
+
1732620732685741 7 13506088516033
\%B19 + \%B19
1387545279120 1387545279120
]

, \%B19, \%B30,
7865521 31 6696179241 25 25769893181 19
\%B19 - \%B19 - \%B19
6006689520 2002229840 49235160
+
1975912990729 13 1048460696489 7 21252634831
\[
\begin{align*}
- & \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \\
& \quad 3003344760 \quad 2002229840 \quad 60066889520 \\
+ \quad 778171189 \quad 31 \quad 1987468196267 \quad 25 \quad 155496778477189 \quad 19 \\
- & \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \\
& \quad 1387545279120 \quad 1387545279120 \quad 693772639560 \\
+ \quad 191631411158401 \quad 13 \quad 300335488637543 \quad 7 \quad 755656533863 \\
- & \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \\
& \quad 693772639560 \quad 1387545279120 \quad 198220754160 \\
+ \quad 1094352947 \quad 31 \quad 2794979430821 \quad 25 \quad 218708802908737 \quad 19 \\
- & \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \\
& \quad 462515093040 \quad 462515093040 \quad 231257645620 \\
+ \quad 91476663003591 \quad 13 \quad 145152550961823 \quad 7 \quad 1564893370717 \\
- & \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \quad - \quad \text{----------} \quad \%B19 \\
& \quad 77085848840 \quad 154171697680 \quad 462515093040 \\
+ \quad 4321823003 \quad 31 \quad 180949646069 \quad 25 \\
- & \quad \%B30 \quad - \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \\
& \quad 1387545279120 \quad 22746643920 \\
+ \quad 863753195062493 \quad 19 \quad 1088094456732317 \quad 13 \\
- & \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \\
& \quad 693772639560 \quad 693772639560 \\
+ \quad 1732620732685741 \quad 7 \quad 1350608516033 \\
- & \quad \text{----------} \quad \%B19 \quad + \quad \text{----------} \quad \%B19 \\
& \quad 1387545279120 \quad 1387545279120 \\
\end{align*}
\]

\[
\begin{align*}
[\%B20, \%B27, \\
& \quad 7869521 \quad 31 \quad 6696179241 \quad 25 \quad 25769893181 \quad 19 \\
- & \quad \text{----------} \quad \%B20 \quad - \quad \text{----------} \quad \%B20 \quad - \quad \text{----------} \quad \%B20 \\
& \quad 6006689520 \quad 2002229840 \quad 492351600 \\
+ \quad 1975912990729 \quad 13 \quad 1048460696489 \quad 7 \quad 21252634831 \\
- & \quad \text{----------} \quad \%B20 \quad - \quad \text{----------} \quad \%B20 \quad - \quad \text{----------} \quad \%B20 \\
& \quad 3003344760 \quad 2002229840 \quad 6006689520 \\
+ \quad 778171189 \quad 31 \quad 1987468196267 \quad 25 \quad 155496778477189 \quad 19 \\
- & \quad \text{----------} \quad \%B20 \quad + \quad \text{----------} \quad \%B20 \quad + \quad \text{----------} \quad \%B20 \\
& \quad 1387545279120 \quad 1387545279120 \quad 693772639560 \\
+ \quad 191631411158401 \quad 13 \quad 300335488637543 \quad 7 \quad 755656533863 \\
- & \quad \text{----------} \quad \%B20 \quad + \quad \text{----------} \quad \%B20 \quad - \quad \text{----------} \quad \%B20 \\
& \quad 693772639560 \quad 1387545279120 \quad 198220754160 \\
+ \quad 1094352947 \quad 31 \quad 2794979430821 \quad 25 \quad 218708802908737 \quad 19
\end{align*}
\]
\[
\begin{array}{cccc}
\text{PACKAGE LEXTRIPK LEXTRIANGULARPACKAGE} & 1385 \\
\hline
62515093040 & 62515093040 & 231257546520 \\
\hline
91476663003591 & 145152550961823 & 7 & 1564893370717 \\
\hline
77085848840 & 154171697680 & 462515093040 \\
\hline
4321823003 & 31 & 180949546609 & 25 \\
\hline
1387545279120 & 22746643920 \\
\hline
863753195062493 & 19 & 1088094456732317 & 13 \\
\hline
693772639560 & 693772639560 \\
\hline
1732620732685741 & 7 & 1350608516033 \\
\hline
1387545279120 & 1387545279120 \\
\hline
7866521 & 31 & 6696179241 & 25 & 2576983181 & 19 \\
\hline
6006669520 & 2002229840 & 49235160 \\
\hline
1975912990729 & 13 & 104846096489 & 7 & 21252634831 \\
\hline
3003344760 & 2002229840 & 6006689520 \\
\hline
778171189 & 31 & 1987468196267 & 25 & 15549671787189 & 19 \\
\hline
1387545279120 & 1387545279120 & 693772639560 \\
\hline
191631411158401 & 13 & 300335488637543 & 7 & 755656433863 \\
\hline
693772639560 & 1387545279120 & 198220754160 \\
\hline
1094352947 & 31 & 2794979430821 & 25 & 21870802908737 & 19 \\
\hline
462515093040 & 462515093040 & 231257546520 \\
\hline
91476663003591 & 145152550961823 & 7 & 1564893370717 \\
\hline
77085848840 & 154171697680 & 462515093040 \\
\hline
4321823003 & 31 & 180949546609 & 25 \\
\hline
1387545279120 & 22746643920 \\
\hline
863753195062493 & 19 & 1088094456732317 & 13 \\
\end{array}
\]
We obtain 24 points given by lists of elements in the RealClosure of Fraction of R. In each list, the first value corresponds to the indeterminate \( f \), the second to \( e \) and so on.

See Also:
- \texttt{help RegularChain}
- \texttt{help RegularTriangularSet}
- \texttt{help SquareFreeRegularTriangularSet}
- \texttt{help ZeroDimensionalSolvePackage}
- \texttt{help NewSparseMultivariatePolynomial}
- \texttt{help LinGroebnerPackage}
- \texttt{help NormalizedTriangularSetCategory}
- \texttt{help RealClosure}
- \texttt{help Fraction}
- \texttt{show LexTriangularPackage}
LexTriangularPackage (LEXTRIPK)

Exports:

fglmIfCan          groebner          lexTriangular
squareFreeLexTriangular  zeroDimensional?  zeroSetSplit

--- package LEXTRIPK LexTriangularPackage ---

)abbrev package LEXTRIPK LexTriangularPackage
++ Author: Marc Moreno Maza
++ Date Created: 08/02/1999
++ Date Last Updated: 08/02/1999
++ References:
++ algebraic towers of simple extensions"
++ Description:
++ A package for solving polynomial systems with finitely many solutions.
++ The decompositions are given by means of regular triangular sets.
++ The computations use lexicographical Groebner bases.
++ The main operations are lexTriangular
++ and squareFreeLexTriangular. The second one provide decompositions by
++ means of square-free regular triangular sets.
++ Both are based on the lexTriangular method described in [1].
++ They differ from the algorithm described in [2] by the fact that
++ multiplicities of the roots are not kept.
++ With the squareFreeLexTriangular operation all multiplicities are removed.
++ With the other operation some multiplicities may remain. Both operations
++ admit an optional argument to produce normalized triangular sets.

LexTriangularPackage(R,ls): Exports == Implementation where

R: GcdDomain
ls: List Symbol
V ==> OrderedVariableList ls
E ==> IndexedExponents V
P ==> NewSparseMultivariatePolynomial(R,V)
TS ==> RegularChain(R,ls)
ST ==> SquareFreeRegularTriangularSet(R,E,V,P)
Q1 ==> Polynomial R
PS ==> GeneralPolynomialSet(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
BWTS ==> Record(val : Boolean, tower : TS)
LpWTS ==> Record(val : (List P), tower : TS)
BWST ==> Record(val : Boolean, tower : ST)
LpWST ==> Record(val : (List P), tower : ST)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppackTS ==> QuasiComponentPackage(R,E,V,P,TS)
regsetgcdpackTS ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)
normalizpackTS ==> NormalizationPackage(R,E,V,P,TS)
quasicomppackST ==> QuasiComponentPackage(R,E,V,P,ST)
regsetgcdpackST ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,ST)
normalizpackST ==> NormalizationPackage(R,E,V,P,ST)

Exports == with

zeroDimensional? : LP -> B
++ \axiom{zeroDimensional?(lp)} returns true iff
++ \axiom{lp} generates a zero-dimensional ideal
++ w.r.t. the variables involved in \axiom{lp}.
fglmIfCan : LP -> Union(LP, "failed")
++ \axiom{fglmIfCan(lp)} returns the lexicographical Groebner
++ basis of \axiom{lp} by using the FGLM strategy,
++ if \axiom{zeroDimensional?(lp)} holds .
groebner : LP -> LP
++ \axiom{groebner(lp)} returns the lexicographical Groebner
++ basis of \axiom{lp}. If \axiom{lp} generates a zero-dimensional
++ ideal then the FGLM strategy is used, otherwise
++ the Sugar strategy is used.
lexTriangular : (LP, B) -> List TS
++ \axiom{lexTriangular(base, norm?)} decomposes the variety
++ associated with \axiom{base} into regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \axiom{base} needs to be a lexicographical Groebner basis
++ of a zero-dimensional ideal. If \axiom{norm?} is \axiom{true}
++ then the regular sets are normalized.
squareFreeLexTriangular : (LP, B) -> List TS
++ \axiom{squareFreeLexTriangular(base, norm?)} decomposes the variety
++ associated with \axiom{base} into square-free regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{base} needs to be a lexicographical Groebner basis
++ of a zero-dimensional ideal. If \texttt{base} is \texttt{true}
++ then the regular sets are normalized.

\texttt{zeroSetSplit: (LP, B) -> List TS}
++ \texttt{zeroSetSplit(lp, \texttt{true})} decomposes the variety
++ associated with \texttt{lp} into regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{lp} needs to generate a zero-dimensional ideal.
++ If \texttt{norm?} is \texttt{true} then the regular sets are normalized.

\texttt{zeroSetSplit: (LP, B) -> List ST}
++ \texttt{zeroSetSplit(lp, \texttt{true})} decomposes the variety
++ associated with \texttt{lp} into square-free regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{lp} needs to generate a zero-dimensional ideal.
++ If \texttt{norm?} is \texttt{true} then the regular sets are normalized.

Implementation == add

\texttt{trueVariables(lp: List(P)): List Symbol ==}
  lv: List V := variables([lp]$PS)
  truels: List Symbol := []
  for s in lv repeat
    if member?(variable(s)::V, lv) then truels := cons(s, truels)
  reverse truels

\texttt{zeroDimensional?(lp:List(P)): Boolean ==}
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  zeroDimensional?(lq1)$fglmmpack

\texttt{fglmIfCan(lp:List(P)): Union(List(P), "failed") ==}
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  foo := fglmIfCan(lq1)$fglmmpack
  foo case "failed" => return("failed" :: Union(List(P), "failed"))
  lp := [retract(q1)$P for q1 in foo :: List(Q1)]
  lp::Union(List(P), "failed")

\texttt{groebner(lp:List(P)): List(P) ==}
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  lq1 := groebner(lq1)$fglmmpack
  lp := [retract(q1)$P for q1 in lq1]
lexTriangular(base: List(P), norm?: Boolean): List(TS) ==
base := sort(infRittWu?,base)
base := remove(zero?, base)
any?(ground?, base) => []
ts: TS := empty()
toSee: List LpWTS := [[base,ts]$LpWTS]
toSave: List TS := []
while not empty? toSee repeat
lpwt := first toSee; toSee := rest toSee
lp := lpwt.val; ts := lpwt.tower
empty? lp => toSave := cons(ts, toSave)
p := first lp; lp := rest lp; v := mvar(p)
algebraic?(v,ts) =>
   error "lexTriangular$LEXTRIPK: should never happen !"
norm? and zero? remainder(init(p),ts).polnum =>
toSee := cons([[lp,ts]$LpWTS, toSee])
(not norm?) and zero? (initiallyReduce(init(p),ts)) =>
toSee := cons([[lp,ts]$LpWTS, toSee])
lbwt: List BWTS := invertible?(init(p),ts)$TS
while (not empty? lbwt) repeat
bwt := first lbwt; lbwt := rest lbwt
b := bwt.val; us := bwt.tower
(not b) => toSee := cons([lp, us], toSee)
lus: List TS
if norm?
   then
      newp := normalizedAssociate(p,us)$normalizpackTS
      lus := [internalAugment(newp,us)$TS]
   else
      newp := p
      lus := augment(newp,us)$TS
newlp := lp
while (not empty? newlp) and (mvar(first newlp) = v) repeat
newlp := rest newlp
for us in lus repeat
   toSee := cons([newlp, us]$LpWTS, toSee)
algebraicSort(toSave)$quasicomppackTS

zeroSetSplit(lp:List(P), norm?:B): List TS ==
bar := fglmIfCan(lp)
bar case "failed" =>
   error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
lexTriangular(bar::(List P),norm?)

squareFreeLexTriangular(base: List(P), norm?: Boolean): List(ST) ==
base := sort(infRittWu?,base)
base := remove(zero?, base)
any?(ground?, base) => []
ts: ST := empty()
toSee: List LpWST := [[base,ts]$LpWST]
toSave: List ST := []
while not empty? toSee repeat
  lpwt := first toSee; toSee := rest toSee
  lp := lpwt.val; ts := lpwt.tower
  empty? lp => toSave := cons(ts, toSave)
  p := first lp; lp := rest lp; v := mvar(p)
  if algebraic?(v,ts) =>
    error "lexTriangular$LEXTRIPK: should never happen!"
  norm? and zero? remainder(init(p),ts).polnum =>
    toSee := cons([[lp, ts]$LpWST, toSee]
      (not norm?) and zero? (initiallyReduce(init(p),ts)) =>
    toSee := cons([[lp, ts]$LpWST, toSee]
  lbwt: List BWST := invertible?(init(p),ts)$ST
  while (not empty? lbwt) repeat
    bwt := first lbwt; lbwt := rest lbwt
    b := bwt.val; us := bwt.tower
    if (not b) => toSee := cons([[lp, us], toSee]
  lus: List ST
  if norm? then
    newp := normalizedAssociate(p,us)$normalizpackST
    lus := augment(newp,us)$ST
  else
    lus := augment(p,us)$ST
  newlp := lp
  while (not empty? newlp) and (mvar(first newlp) = v) repeat
    newlp := rest newlp
  for us in lus repeat
    toSee := cons([[newlp, us]$LpWST, toSee]
  algebraicSort(toSave)$quasicomppackST

zeroSetSplit(lp:List(P), norm?:B): List ST ==
  bar := fglmIfCan(lp)
  bar case "failed" =>
    error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
  squareFreeLexTriangular(bar::(List P),norm?)

——

— LEXTRIPK.dotab —

"LEXTRIPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LEXTRIPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"LEXTRIPK" -> "SFRTCAT"
package LINDEP LinearDependence

— LinearDependence.input —

)set break resume
)sys rm -f LinearDependence.output
)spool LinearDependence.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearDependence
--E 1

)spool
)lisp (bye)

— LinearDependence.help —

====================================================================
LinearDependence examples
====================================================================

Test for linear dependence.

See Also:
o )show LinearDependence

———
LinearDependence (LINDEP)

Exports:

linearDependence  linearlyDependent?  solveLinear

— package LINDEP LinearDependence —

)abbrev package LINDEP LinearDependence
++ Author: Manuel Bronstein
++ Date Last Updated: 14 May 1991
++ Description:
++ Test for linear dependence.

LinearDependence(S, R):Exports == Implementation where
  S: IntegralDomain
  R: LinearlyExplicitRingOver S
  Q ==> Fraction S

Exports ==> with
  linearlyDependent?: Vector R -> Boolean
    ++ \spad{linearlyDependent?([v1,...,vn])} returns true if
    ++ the vi's are linearly dependent over S, false otherwise.
  linearDependence : Vector R -> Union(Vector S, "failed")
    ++ \spad{linearDependence([v1,...,vn])} returns \spad{[c1,...,cn]} if
    ++ \spad{c1*v1 + ... + cn*vn = 0} and not all the ci's are 0,
    ++ "failed" if the vi's are linearly independent over S.
  if S has Field then
    solveLinear: (Vector R, R) -> Union(Vector S, "failed")
      ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
      ++ such that \spad{c1*v1 + ... + cn*vn = u},
      ++ "failed" if no such ci's exist in S.
  else
    solveLinear: (Vector R, R) -> Union(Vector Q, "failed")
      ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
      ++ such that \spad{c1*v1 + ... + cn*vn = u},
      ++ "failed" if no such ci's exist in the quotient field of S.
Implementation ==> add

aNonZeroSolution: Matrix S -> Union(Vector S, "failed")

aNonZeroSolution m ==
  every?(zero?, v := first nullSpace m) => "failed"
  v

linearlyDependent? v ==
  zero?(n := #v) => true
  one? n => zero?(v(minIndex v))
  (n = 1) => zero?(v(minIndex v))
  positive? nullity reducedSystem transpose v

linearDependence v ==
  zero?(n := #v) => empty()
  one? n =>
    (n = 1) =>
      zero?(v(minIndex v)) => new(1, 1)
      "failed"
      aNonZeroSolution reducedSystem transpose v

if S has Field then
  solveLinear(v:Vector R, c:R):Union(Vector S, "failed") ==
    zero? c => new(#v, 0)
    empty? v => "failed"
    sys := reducedSystem(transpose v, new(1, c))
    particularSolution(sys.mat, sys.vec)$LinearSystemMatrixPackage(S, Vector S, Vector S, Matrix S)
  else
    solveLinear(v:Vector R, c:R):Union(Vector Q, "failed") ==
    zero? c => new(#v, 0)
    empty? v => "failed"
    sys := reducedSystem(transpose v, new(1, c))
    particularSolution(map((z:S):Q+->z::Q, sys.mat),
                      $MatrixCategoryFunctions2(S, Vector S, Vector S, Matrix S, Q, Vector Q, Vector Q, Matrix Q),
                      map((z1:S):Q+->z1::Q, sys.vec)$VectorFunctions2(S, Q))$LinearSystemMatrixPackage(Q, Vector Q, Vector Q, Matrix Q)

— LINDEP.dotabb —

"LINDEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LINDEP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LINDEP" -> "PFECAT"

package LODOF LinearOrdinaryDifferentialOperatorFactorizer

-- LinearOrdinaryDifferentialOperatorFactorizer.input --

)set break resume
)sys rm -f LinearOrdinaryDifferentialOperatorFactorizer.output
)spool LinearOrdinaryDifferentialOperatorFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearOrdinaryDifferentialOperatorFactorizer
--E 1

)spool
)lisp (bye)

-- LinearOrdinaryDifferentialOperatorFactorizer.help --

====================================================================
LinearOrdinaryDifferentialOperatorFactorizer examples
====================================================================

LinearOrdinaryDifferentialOperatorFactorizer provides a factorizer for
linear ordinary differential operators whose coefficients are rational
functions.

See Also:
o )show LinearOrdinaryDifferentialOperatorFactorizer
LinearOrdinaryDifferentialOperatorFactorizer (LODOF)

Exports:
  factor   factor1

— package LODOF LinearOrdinaryDifferentialOperatorFactorizer —

)abbrev package LODOF LinearOrdinaryDifferentialOperatorFactorizer
++ Author: Fritz Schwarz, Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{LinearOrdinaryDifferentialOperatorFactorizer} provides a
++ factorizer for linear ordinary differential operators whose coefficients
++ are rational functions.

LinearOrdinaryDifferentialOperatorFactorizer(F, UP): Exports == Impl where
  F : Join(Field, CharacteristicZero,
        RetractableTo Integer, RetractableTo Fraction Integer)
  UP: UnivariatePolynomialCategory F
  RF ==> Fraction UP
  L ==> LinearOrdinaryDifferentialOperator1 RF

Exports ==
with
  factor: (L, UP -> List F) -> List L
    ++ factor(a, zeros) returns the factorisation of a.
    ++ \spad{zeros} is a zero finder in \spad{UP}.
  if F has AlgebraicallyClosedField then
    factor: L -> List L
      ++ factor(a) returns the factorisation of a.
    factor1: L -> List L
      ++ factor1(a) returns the factorisation of a,
      ++ assuming that a has no first-order right factor.

Impl ==
  add
  import RationalLODE(F, UP)
import RationalRicDE(F, UP)
-- import AssociatedEquations RF

definitions

definitions

expsol : (L, UP -> List F, UP -> Factored UP) -> Union(RF, "failed")
expsol : (L, UP -> List F, UP -> Factored UP, Boolean) -> List RF
opval : (L, L) -> L
rfactor : (L, L, UP -> List F, UP -> Factored UP, Boolean) -> List L
rightFactor : (L, NonNegativeInteger, UP -> List F, UP -> Factored UP) -> Union(L, "failed")
innerFactor : (L, UP -> List F, UP -> Factored UP, Boolean) -> List L

factor(l, zeros) == innerFactor(l, zeros, squareFree, true)

expsol(l, zeros, ezfactor) ==
  empty?(sol := expsolsl(l, zeros, ezfactor, false)) => "failed"
  first sol

expsol(l, zeros, ezfactor, all?) ==
  sol := [differentiate(f)/f for f in ratDsolve(l, 0).basis | f ^= 0]
  not(all? or empty? sol) => sol
  concat(sol, ricDsolve(l, zeros, ezfactor))

-- opval(l1, l2) returns l1(l2)
opval(l1, l2) ==
  ans:L := 0
  l2n:L := 1
  for i in 0..degree l1 repeat
    ans := ans + coefficient(l1, i) * l2n
  l2n := l2 * l2n

-- r1? is true means look for 1st-order right-factor also
innerFactor(l, zeros, ezfactor, r1?) ==
  (n := degree l) <= 1 => [l]
  l1 := adjoint l
for i in 1..(n quo 2) repeat
  (r1? or (i > 1)) and ((u := rightFactor(l, i, zeros, ezfactor)) case L) =>
    return concat_!(rfactor(l, u::L, zeros, ezfactor, false), u::L)
  (2 * i < n) and ((u := rightFactor(ll, i, zeros, ezfactor)) case L) =>
    return concat(adjoint(u::L), rfactor(ll, u::L, zeros, ezfactor, true))

rightFactor(l, n, zeros, ezfactor) ==
  -- one? n =>
  (n = 1) =>
    (u := expsol(l, zeros, ezfactor)) case "failed" => "failed"
    D() - u::RF::L
  -- rec := associatedEquations(l, n::PositiveInteger)
  -- empty?(sol := expsols(rec.eq, zeros, ezfactor, true)) => "failed"
  "failed"

if F has AlgebraicallyClosedField then
  zro1: UP -> List F
  zro : (UP, UP -> Factored UP) -> List F

  zro(p, ezfactor) ==
    concat [zro1(r.factor) for r in factors ezfactor p]
  zro1 p ==
    [zeroOf(map((z1:F):F+->z1,p)_$UnivariatePolynomialCategoryFunctions2(F, UP, F, SparseUnivariatePolynomial F))]

if F is AlgebraicNumber then
  import AlgFactor UP

  factor l  ==
    innerFactor(l, (p:UP):List(F)+->zro(p,factor),factor,true)
  factor1 l ==
    innerFactor(l, (p:UP):List(F)+->zro(p,factor),factor,false)

else
  factor l  ==
    innerFactor(l, (p:UP):List(F)+->zro(p,squareFree),squareFree,true)
  factor1 l ==
    innerFactor(l, (p:UP):List(F)+->zro(p,squareFree),squareFree,false)

"LODOF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
package LODOOPS LinearOrdinaryDifferentialOperatorsOps

LinearOrdinaryDifferentialOperatorsOps.input

)set break resume
)sys rm -f LinearOrdinaryDifferentialOperatorsOps.output
)spool LinearOrdinaryDifferentialOperatorsOps.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearOrdinaryDifferentialOperatorsOps
--E 1

)spool
)lisp (bye)

LinearOrdinaryDifferentialOperatorsOps.help

====================================================================
LinearOrdinaryDifferentialOperatorsOps examples
====================================================================

LinearOrdinaryDifferentialOperatorsOps provides symmetric products and sums for linear ordinary differential operators.

See Also:
  o )show LinearOrdinaryDifferentialOperatorsOps
LinearOrdinaryDifferentialOperatorsOps (LODOOPS)

Exports:

directSum  symmetricPower  symmetricProduct

--- package LODOOPS LinearOrdinaryDifferentialOperatorsOps ---

)abbrev package LODOOPS LinearOrdinaryDifferentialOperatorsOps
++ Author: Manuel Bronstein
++ Date Created: 18 January 1994
++ Date Last Updated: 15 April 1994
++ Description:
++ \spad{LinearOrdinaryDifferentialOperatorsOps} provides symmetric
++ products and sums for linear ordinary differential operators.
-- Putting those operations here rather than defaults in LODOCAT allows
-- LODOCAT to be defined independently of the derivative used.
-- MB 1/94

LinearOrdinaryDifferentialOperatorsOps(A, L): Exports == Implementation where
A: Field
L: LinearOrdinaryDifferentialOperatorCategory A

N ==> NonNegativeInteger
V ==> OrderlyDifferentialVariable Symbol
P ==> DifferentialSparseMultivariatePolynomial(A, Symbol, V)

Exports == with

symmetricProduct: (L, L, A -> A) -> L
  ++ symmetricProduct(a,b,D) computes an operator \spad{c} of
  ++ minimal order such that the nullspace of \spad{c} is
  ++ generated by all the products of a solution of \spad{a} by
  ++ a solution of \spad{b}.
  ++ D is the derivation to use.
symmetricPower: (L, N, A -> A) -> L
  ++ symmetricPower(a,n,D) computes an operator \spad{c} of
  ++ minimal order such that the nullspace of \spad{c} is
  ++ generated by all the products of \spad{n} solutions
++ of \texttt{a}.
++ \texttt{D} is the derivation to use.

\texttt{directSum: (L, L, A \rightarrow A) \rightarrow L}
++ \texttt{directSum(a,b,D)} computes an operator \texttt{c} of
++ minimal order such that the nullspace of \texttt{c} is
++ generated by all the sums of a solution of \texttt{a} by
++ a solution of \texttt{b}.
++ \texttt{D} is the derivation to use.

Implementation ==> add

\texttt{import IntegerCombinatoricFunctions}

\texttt{var1 := new()$Symbol}
\texttt{var2 := new()$Symbol}

\texttt{nonTrivial?: Vector A \rightarrow Boolean}
\texttt{applyLODO : (L, V) \rightarrow P}
\texttt{killer : (P, N, List V, List P, A \rightarrow A) \rightarrow L}
\texttt{vec2LODO : Vector A \rightarrow L}

\texttt{nonTrivial? v == any?((x1:A):Boolean +-> x1 \neq 0, v)$Vector(A)}
\texttt{vec2LODO v == +/[monomial(v.i, (i-1)::N) for i in 1..#v]}

\texttt{symmetricPower(l, m, diff) ==}
\texttt{u := var1::V; n := degree l}
\texttt{un := differentiate(u, n)}
\texttt{a := applyLODO(inv(- leadingCoefficient l) \ast reductum l, u)}
\texttt{killer(u::P ** m, binomial(n + m - 1, n - 1)::N, [un], [a], diff)}

\texttt{symmetricProduct(l1, l2, diff) ==}
\texttt{u := var1::V; v := var2::V}

-- returns an operator L such that L(u) = 0, for a given differential
-- polynomial u, given that the differential variables appearing in u
-- satisfy some linear ode’s
-- \texttt{m} is a bound on the order of the operator searched.
-- \texttt{lvar, lval} describe the substitution(s) to perform when differentiating
-- the expression \texttt{u} \texttt{(}they encode the fact the the differential variables
-- satisfy some differential equations, which can be seen as the rewrite
-- rules \texttt{lvar --> lval)}
-- \texttt{diff} is the derivation to use
\texttt{killer(u, m, lvar, lval, diff) ==}
\texttt{lu:List P := [u]}
for q in 0..m repeat
\texttt{mat := reducedSystem(matrix([lu])@Matrix(P))@Matrix(A)}
\texttt{(sol := find(nonTrivial?, 1 := nullSpace mat)) case Vector(A) =>}
\texttt{return vec2LODO(sol::Vector(A))}
\texttt{u := eval(differentiate(u, diff), lvar, lval)}
\texttt{lu := concat_!(lu, [u])}
\texttt{error "killer: no linear dependence found"}

\texttt{symmetricProduct(l1, l2, diff) ==}
\texttt{u := var1::V; v := var2::V}
n1 := degree l1; n2 := degree l2
un := differentiate(u, n1); vn := differentiate(v, n2)
a := applyLADO(inv(- leadingCoefficient l1) * reductum l1, u)
b := applyLADO(inv(- leadingCoefficient l2) * reductum l2, v)
killer(u::P * v::P, n1 * n2, [un, vn], [a, b], diff)

directSum(l1, l2, diff) ==
  u := var1::V; v := var2::V
  n1 := degree l1; n2 := degree l2
  un := differentiate(u, n1); vn := differentiate(v, n2)
  a := applyLADO(inv(- leadingCoefficient l1) * reductum l1, u)
  b := applyLADO(inv(- leadingCoefficient l2) * reductum l2, v)
  killer(u::P + v::P, n1 + n2, [un, vn], [a, b], diff)

applyLADO(l, v) ==
  p::P := 0
  while l ^= 0 repeat
    p := p + monomial(leadingCoefficient(l)::P,
                      differentiate(v, degree l), 1)
    l := reductum l
  P

— LODOOPS.dotabb —

"LODOOPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOOPS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LODOOPS" -> "ALIST"

package LPEFRAC LinearPolynomialEquationByFractions

— LinearPolynomialEquationByFractions.input —

)set break resume
)sys rm -f LinearPolynomialEquationByFractions.output
)spool LinearPolynomialEquationByFractions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1


LinearPolynomialEquationByFractions (LPEFRAC)

Exports:
solveLinearPolynomialEquationByFractions

—— package LPEFRAC LinearPolynomialEquationByFractions ——

)abbrev package LPEFRAC LinearPolynomialEquationByFractions
++ Author: James Davenport
++ Description:
++ Given a PolynomialFactorizationExplicit ring, this package
++ provides a defaulting rule for the \spad{solveLinearPolynomialEquation}
++ operation, by moving into the field of fractions, and solving it there
++ via the \spad{multiEuclidean} operation.

\begin{verbatim}
LinearPolynomialEquationByFractions(R:PolynomialFactorizationExplicit): with
  solveLinearPolynomialEquationByFractions: ( _
    List SparseUnivariatePolynomial R, _
    SparseUnivariatePolynomial R) -> _
    Union(List SparseUnivariatePolynomial R, "failed")
++ solveLinearPolynomialEquationByFractions([f1, ..., fn], g)
++ (where the fi are relatively prime to each other)
++ returns a list of ai such that
++ \spad{g/prod fi = sum ai/fi}
++ or returns "failed" if no such exists.

== add
SupR ==> SparseUnivariatePolynomial R
F ==> Fraction R
SupF ==> SparseUnivariatePolynomial F
import UnivariatePolynomialCategoryFunctions2(R,SupR,F,SupF)
lp : List SupR
pp: SupR
pF: SupF
pullback : SupF -> Union(SupR,"failed")
pullback(pF) ==
  pF = 0 => 0
  c:=retractIfCan leadingCoefficient pF
  c case "failed" => "failed"
  r:=pullback reductum pF
  r case "failed" => "failed"
  monomial(c,degree pF) + r
solveLinearPolynomialEquationByFractions(lp,pp) ==
  lpF:List SupF:=[map((x:R):F +-> x@R::F,u) for u in lp]
pF:SupF:=map((x:R):F ++> x::F,pp)
ans:= solveLinearPolynomialEquation(lpF,pF)$F
ans case "failed" => "failed"
[(vv:= pullback v;
    vv case "failed" => return "failed";
    vv)
    for v in ans]
\end{verbatim}

— LPEFRAC.dotabb —

"LPEFRAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LPEFRAC"]
package LISYSER LinearSystemFromPowerSeriesPackage

— LinearSystemFromPowerSeriesPackage.input —

)set break resume
)sys rm -f LinearSystemFromPowerSeriesPackage.output
)spool LinearSystemFromPowerSeriesPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemFromPowerSeriesPackage
--R
--R LinearSystemFromPowerSeriesPackage(K: Field,PCS: LocalPowerSeriesCategory(K)) is a package
--R Abbreviation for LinearSystemFromPowerSeriesPackage is LISYSER
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LISYSER
--R
--R-------------------------------- Operations --------------------------------
--R finiteSeries2LinSys : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2LinSysWOVectorise : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2Vector : (PCS,Integer) -> List(K)
--R
--E 1

)spool
)lisp (bye)

— LinearSystemFromPowerSeriesPackage.help —

==============================================
LinearSystemFromPowerSeriesPackage examples
==============================================

Part of the PAFF package

See Also:
o )show LinearSystemFromPowerSeriesPackage

LinearSystemFromPowerSeriesPackage (LISYSER)

Exports:
finiteSeries2LinSys finiteSeries2LinSysWOVectorise finiteSeries2Vector

--- package LISYSER LinearSystemFromPowerSeriesPackage ---

)abbrev package LISYSER LinearSystemFromPowerSeriesPackage
++ Authors: Gaetan Hache
++ Date Created: 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ Part of the PAFF package
LinearSystemFromPowerSeriesPackage(K,PCS):P==T where
  K : Field
  PCS: LocalPowerSeriesCategory(K)

INT ==> Integer
TERM ==> Record(k:INT,c:K)
SER ==> Stream(TERM)
LOpPack ==> LinesOpPack(K)

P==> with
  finiteSeries2LinSysWOVectorise: (List PCS, INT) -> Matrix K

finiteSeries2LinSys: (List PCS, INT) -> Matrix K
  ++ finiteSeries2LinSys(ls,n) returns a matrix which right kernel
  ++ is the solution of the linear combinations of the series in ls
  ++ which has order greater or equal to n.
++ NOTE: All the series in ls must be finite and must have order
++ at least 0: so one must first call on each of them the
++ function filterUpTo(s,n) and apply an appropriate shift
++ (mult by a power of t).

finiteSeries2Vector: (PCS, INT) -> List K

T==> add

finiteSeries2ListOfTerms: PCS -> List TERM

finiteSeries2ListOfTermsStream: SER -> List TERM

finiteSeries2ListOfTermsStream(s)==
  empty?(s) => empty()
  cons(frst s, finiteSeries2ListOfTermsStream(rst(s)))

finiteSeries2LinSys(ls,n)==
  ll:List K:= [0$K]
  lZero:=new(#ls pretend NonNegativeInteger,11)$List(List(K))
  n <= 0 => transpose matrix lZero
  tMat:=transpose matrix [finiteSeries2Vector(s,n) for s in ls]
  rowEchNoZeroLines(tMat)$LOpPack

finiteSeries2LinSysWOVectorise(ls,n)==
  ll:List K:= [0$K]
  lZero:=new(#ls pretend NonNegativeInteger,11)$List(List(K))
  n <= 0 => transpose matrix lZero
  tMat:=transpose matrix [finiteSeries2Vector(s,n) for s in ls]
  rowEchNoZeroLinesWOVectorise(tMat)$LOpPack

finiteSeries2ListOfTerms(s)==
  ss:SER:= s :: SER
  finiteSeries2ListOfTermsStream(ss)

finiteSeries2Vector(ins,n)==
  lZero:=new((n pretend NonNegativeInteger),0)$List(K)
  s:= removeFirstZeroes ins
  10fTerm:=finiteSeries2ListOfTermsStream(s)
  for t in 10fTerm repeat lZero.((t.k)+1):= t.c
  lZero

— LYSYER.dotabb —

"LYSER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LYSER"]
"LOCP0WC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LOCP0WC"]
"LISYSER" -> "LOCPOWC"

package LSMP LinearSystemMatrixPackage

— LinearSystemMatrixPackage.input —

)set break resume
)sys rm -f LinearSystemMatrixPackage.output
)spool LinearSystemMatrixPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemMatrixPackage
--E 1

)spool
)lisp (bye)

— LinearSystemMatrixPackage.help —

====================================================================
LinearSystemMatrixPackage examples
====================================================================

This package solves linear system in the matrix form AX = B.

See Also:
o )show LinearSystemMatrixPackage
LinearSystemMatrixPackage (LSMP)

Exports:
  hasSolution?   particularSolution   rank   solve

--- package LSMP LinearSystemMatrixPackage ---

)abbrev package LSMP LinearSystemMatrixPackage
++ Author: P.Gianni, S.Watt
++ Date Created: Summer 1985
++ Date Last Updated: Summer 1990
++ Description:
++ This package solves linear system in the matrix form \spad{AX = B}.

LinearSystemMatrixPackage(F, Row, Col, M): Cat == Capsule where
  F: Field
  Row: FiniteLinearAggregate F with shallowlyMutable
  Col: FiniteLinearAggregate F with shallowlyMutable
  M : MatrixCategory(F, Row, Col)

N     ==> NonNegativeInteger
PartialV ==> Union(Col, "failed")
Both   ==> Record(particular: PartialV, basis: List Col)

Cat ==> with
  solve  : (M, Col) -> Both
    ++ solve(A,B) finds a particular solution of the system \spad{AX = B}
    ++ and a basis of the associated homogeneous system \spad{AX = 0}.
  solve  : (M, List Col) -> List Both
    ++ solve(A,LB) finds a particular soln of the systems \spad{AX = B}
    ++ and a basis of the associated homogeneous systems \spad{AX = 0}
    ++ where B varies in the list of column vectors LB.

  particularSolution: (M, Col) -> PartialV
    ++ particularSolution(A,B) finds a particular solution of the linear
    ++ system \spad{AX = B}.
  hasSolution?: (M, Col) -> Boolean
++ hasSolution?(A,B) tests if the linear system \(AX = B\) has a solution.
rank : (M, Col) -> N
++ rank(A,B) computes the rank of the complete matrix \(\begin{pmatrix} A \mid B \end{pmatrix}\) of the linear system \(AX = B\).

Capsule ==> add
systemMatrix : (M, Col) -> M
aSolution : M -> PartialV
-- rank theorem
hasSolution?(A, b) == rank A = rank systemMatrix(A, b)
systemMatrix(m, v) == horizConcat(m, -(v::M))
rank(A, b) == rank systemMatrix(A, b)
particularSolution(A, b) == aSolution rowEchelon systemMatrix(A,b)

-- m should be in row-echelon form.
-- last column of m is -(right-hand-side of system)
aSolution m ==
nvar := (ncols m - 1)::N
rk := maxRowIndex m
while (rk >= minRowIndex m) and every?(zero?, row(m, rk))
  repeat rk := dec rk
rk < minRowIndex m => new(nvar, 0)
ck := minColIndex m
while (ck < maxColIndex m) and zero? qelt(m, rk, ck) repeat
  ck := inc ck
ck = maxColIndex m => "failed"
sol := new(nvar, 0)$Col
-- find leading elements of diagonal
v := new(nvar, minRowIndex m - 1)$PrimitiveArray(Integer)
for i in minRowIndex m .. rk repeat
  for j in 0.. while zero? qelt(m, i, j+minColIndex m) repeat 0
    v.j := i
  for j in 0..nvar-1 repeat
    if v.j >= minRowIndex m then
      qsetelt_!(sol, j+minIndex sol, - qelt(m, v.j, maxColIndex m))
sol

solve(A:M, b:Col) ==
-- Special case for homogeneous systems.
every?(zero?, b) => [new(ncols A, 0), nullSpace A]
-- General case.
m := rowEchelon systemMatrix(A, b)
[aSolution m,
 nullSpace subMatrix(m, minRowIndex m, maxRowIndex m,
                 minColIndex m, maxColIndex m - 1)]
solve(A:M, l:List Col) ==
null l => [[new(ncols A, 0), nullSpace A]]
nl := (sol0 := solve(A, first l)).basis
cons(sol0,
    [[aSolution rowEchelon systemMatrix(A, b), nl]
    for b in rest l])

---

--- LSMP.dotabb ---

"LSMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LSMP"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"LSMP" -> "MATCAT"

---

package LSMP1 LinearSystemMatrixPackage1

--- LinearSystemMatrixPackage1.input ---

)set break resume
)sys rm -f LinearSystemMatrixPackage1.output
)spool LinearSystemMatrixPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemMatrixPackage1
--E 1

)spool
)lisp (bye)

---

--- LinearSystemMatrixPackage1.help ---

====================================================================
LinearSystemMatrixPackage1 examples
====================================================================

This package solves linear system in the matrix form AX = B. It is
essentially a particular instantiation of the package
LinearSystemMatrixPackage for Matrix and Vector. This package's existence makes it easier to use solve in the AXIOM interpreter.

See Also:
o )show LinearSystemMatrixPackage1

LinearSystemMatrixPackage1 (LSMP1)

Exports:
hasSolution? particularSolution rank solve

— package LSMP1 LinearSystemMatrixPackage1 —

)abbrev package LSMP1 LinearSystemMatrixPackage1
++ Author: R. Sutor
++ Date Created: June, 1994
++ Description:
++ This package solves linear system in the matrix form \spad{AX = B}.
++ It is essentially a particular instantiation of the package
++ \spadtype{LinearSystemMatrixPackage} for Matrix and Vector. This
++ package's existence makes it easier to use \spadfun{solve} in the
++ AXIOM interpreter.

LinearSystemMatrixPackage1(F): Cat == Capsule where
  F: Field
  Row ==> Vector F
  Col ==> Vector F
  M ==> Matrix(F)
  LL ==> List List F
  N ==> NonNegativeInteger
PartialV ==> Union(Col, "failed")
Both  ==> Record(particular: PartialV, basis: List Col)
LSMP  ==> LinearSystemMatrixPackage(F, Row, Col, M)

Cat ==> with
solve : (M, Col) -> Both
  ++ solve(A,B) finds a particular solution of the system \(AX = B\)
  ++ and a basis of the associated homogeneous system \(AX = 0\).
solve : (LL, Col) -> Both
  ++ solve(A,B) finds a particular solution of the system \(AX = B\)
  ++ and a basis of the associated homogeneous system \(AX = 0\).
solve : (M, List Col) -> List Both
  ++ solve(A,LB) finds a particular soln of the systems \(AX = B\)
  ++ and a basis of the associated homogeneous systems \(AX = 0\)
  ++ where B varies in the list of column vectors LB.
solve : (LL, List Col) -> List Both
  ++ solve(A,LB) finds a particular soln of the systems \(AX = B\)
  ++ and a basis of the associated homogeneous systems \(AX = 0\)
  ++ where B varies in the list of column vectors LB.

particularSolution: (M, Col) -> PartialV
  ++ particularSolution(A,B) finds a particular solution of the linear
  ++ system \(AX = B\).
hasSolution?: (M, Col) -> Boolean
  ++ hasSolution?(A,B) tests if the linear system \(AX = B\)
  ++ has a solution.
rank : (M, Col) -> N
  ++ rank(A,B) computes the rank of the complete matrix \(AX = B\).

Capsule ==> add
solve(m : M, c: Col): Both == solve(m,c)$LSMP
solve(ll : LL, c: Col): Both == solve(matrix(ll)$M,c)$LSMP
solve(m : M, l : List Col): List Both == solve(m, l)$LSMP
solve(ll : LL, l : List Col): List Both == solve(matrix(ll)$M, l)$LSMP
particularSolution (m : M, c : Col): PartialV == particularSolution(m, c)$LSMP
hasSolution?(m :M, c : Col): Boolean == hasSolution?(m, c)$LSMP
rank(m : M, c : Col): N == rank(m, c)$LSMP
package LSPP LinearSystemPolynomialPackage

--- LinearSystemPolynomialPackage.input ---

)set break resume
)sys rm -f LinearSystemPolynomialPackage.output
)spool LinearSystemPolynomialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemPolynomialPackage
--E 1

)spool
)lisp (bye)

---

--- LinearSystemPolynomialPackage.help ---

====================================================================
LinearSystemPolynomialPackage examples
====================================================================

This package finds the solutions of linear systems presented as a list of polynomials.

See Also:
  o )show LinearSystemPolynomialPackage

---
LinearSystemPolynomialPackage (LSPP)

Exports:
linSolve

--- package LSPP LinearSystemPolynomialPackage ---

)abbrev package LSPP LinearSystemPolynomialPackage
++ Author: P.Gianni
++ Date Created: Summer 1985
++ Date Last Updated: Summer 1993
++ Description:
++ This package finds the solutions of linear systems presented as a
++ list of polynomials.

LinearSystemPolynomialPackage(R, E, OV, P): Cat == Capsule where
  R : IntegralDomain
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)

  F ==> Fraction P
  NNI ==> NonNegativeInteger
  V ==> Vector
  M ==> Matrix
  Soln ==> Record(particular: Union(V F, "failed"), basis: List V F)

Cat == with
  linSolve: (List P, List OV) -> Soln
  ++ linSolve(lp,lvar) finds the solutions of the linear system
  ++ of polynomials lp = 0 with respect to the list of symbols lvar.

Capsule == add

    ---- Local Functions ----

  poly2vect: (P, List OV) -> Record(coefvec: V F, reductum: F)
intoMatrix: (List P, List OV) -> Record(mat: M F, vec: V F)

poly2vect(p : P, vs : List OV) : Record(coefvec: V F, reductum: F) ==
coefs := new(#vs, 0)$(V F)
for v in vs for i in 1.. while p ^= 0 repeat
  u := univariate(p, v)
  degree u = 0 => "next v"
  coefs.i := (c := leadingCoefficient u)::F
  p := p - monomial(c,v, 1)
[coefs, p :: F]

intoMatrix(ps : List P, vs : List OV) : Record(mat: M F, vec: V F) ==
m := zero(#ps, #vs)$M(F)
v := new(#ps, 0)$V(F)
for p in ps for i in 1.. repeat
  totalDegree(p,vs) > 1 => error "The system is not linear"
  r := poly2vect(p,vs)
  m:=setRow_!(m,i,r.coefvec)
  v.i := - r.reductum
[m, v]

linSolve(ps, vs) ==
r := intoMatrix(ps, vs)
solve(r.mat, r.vec)$LinearSystemMatrixPackage(F,V F,V F,M F)

---

— LSPP.dotabb —

"LSPP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LSPP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LSPP" -> "PFECAT"

---

package LGROBP LinGroebnerPackage

— LinGroebnerPackage.input —

)set break resume
)sys rm -f LinGroebnerPackage.output
)spool LinGroebnerPackage.output
)set message test on
Given a Groebner basis \( B \) with respect to the total degree ordering for a zero-dimensional ideal \( I \), compute a Groebner basis with respect to the lexicographical ordering by using linear algebra.

See Also:
- \( \)show LinGroebnerPackage

---

**LinGroebnerPackage (LGROBP)**

Exports:
- anticoord
- choosemon
- computeBasis
- coord
- groebgen
- intcompBasis
- linGenPos
- minPol
- totolex
- transform

---

---

---

---
)abbrev package LGROBP LinGroebnerPackage
++ Description:
++ Given a Groebner basis B with respect to the total degree ordering for
++ a zero-dimensional ideal I, compute
++ a Groebner basis with respect to the lexicographical ordering by using
++ linear algebra.

LinGroebnerPackage(lv,F) : C == T

where
Z ==> Integer
lv : List Symbol
F : GcdDomain
DP ==> DirectProduct(#lv,NonNegativeInteger)
DPoly ==> DistributedMultivariatePolynomial(lv,F)
HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,F)
OV ==> OrderedVariableList(lv)
NNI ==> NonNegativeInteger
LVals ==> Record(gblist : List DPoly,gvlist : List Z)
VF ==> Vector F
VV ==> Vector NNI
MF ==> Matrix F
cLVars ==> Record(glbase:List DPoly,glval:List Z)

C == with

linGenPos : List HDPoly -> LVals
++ linGenPos undocumented
groebgen : List DPoly -> cLVars
++ groebgen undocumented
totolex : List HDPoly -> List DPoly
++ totolex undocumented
minPol : (List HDPoly,List HDPoly,OV) -> HDPoly
++ minPol undocumented
minPol : (List HDPoly,OV) -> HDPoly
++ minPol undocumented

computeBasis : List HDPoly -> List HDPoly
++ computeBasis undocumented
coord : (HDPoly,List HDPoly) -> VF
++ coord undocumented
anticoord : (List F,DPoly,List DPoly) -> DPoly
++ anticoord undocumented
intcompBasis : (OV,List HDPoly,List HDPoly) -> List HDPoly
++ intcompBasis undocumented
choosemon : (DPoly, List DPoly) -> DPoly
++ choosemon \undocumented

transform : DPoly -> HDPoly
++ transform \undocumented

T == add

import GroebnerPackage(F,DP,OV,DPoly)
import GroebnerPackage(F,HDP,OV,HDPoly)
import GroebnerInternalPackage(F,HDP,OV,HDPoly)
import GroebnerInternalPackage(F,DP,OV,DPoly)

lvar := [variable(yx)::OV for yx in lv]

reduceRow(M:MF, v : VF, lastRow: Integer, pivots: Vector(Integer)) : VF ==
a1:F := 1
b:F := 0
dim := #v
for j in 1..lastRow repeat -- scan over rows
  mj := row(M,j)
  k := pivots(j)
  b := mj.k
  vk := v.k
  for kk in 1..(k-1) repeat
    v(kk) := ((-b*v(kk)) exquo a1) :: F
  for kk in k..dim repeat
    v(kk) := ((vk*mj(kk)-b*v(kk)) exquo a1)::F
  a1 := b
v

rRedPol(f:HDPoly, B:List HDPoly):Record(poly:HDPoly, mult:F) ==
gm := redPo(f,B)
gm.poly = 0 => gm
gg := reductum(gm.poly)
ggm := rRedPol(gg,B)
[ggm.mult*(gm.poly - gg) + ggm.poly, ggm.mult*gm.mult]

----- transform the total basis B in lex basis -----
totolex(B : List HD Poly) : List DPoly ==
result:List DPoly :=[]
ltresult:List DPoly :=[]
vBasis:= computeBasis B
nBasis:List DPoly :=[1$DPoly]
ndim:=(#vBasis)::PositiveInteger
ndim1:NNI:=ndim+1
lm:VF
linmat:MF:=zero(ndim,2*ndim+1)
linmat(1,1):=1$F
linmat(1,ndim1):=1
pivots: Vector Integer := new(ndim,0)
pivots(1) := 1
firstmon: DPoly := 1$DPoly
ofirstmon: DPoly := 1$DPoly
orecfmon: Record(poly: HDPoly, mult: F) := [1,1]
i: NNI := 2
while (firstmon := choosemon(firstmon, ltresult)) ^= 1 repeat
  if (v := firstmon exquo ofirstmon) case "failed" then
    recfmon := rRedPol(transform firstmon, B)
  else
    v := firstmon exquo ofirstmon
    recfmon := rRedPol(transform(v$DPoly) * orecfmon.poly, B)
    recfmon.mult := recfmon.mult * orecfmon.mult
    cc := gcd(content recfmon.poly, recfmon.mult)
    recfmon.poly := (recfmon.poly exquo cc)::HDPoly
    recfmon.mult := (recfmon.mult exquo cc)::F
    vBasis := concat(orecfmon.poly, vBasis)
    orecfmon := recfmon
  if i = ndim1 then j := ndim1
  else
    j := 1
    while lm(j) = 0 and j < ndim1 repeat j := j + 1
    if j = ndim1 then
      cordlist: List F := [lm(k) for k in ndim1..ndim1+(*nBasis)]
      antc := +[c*b for c in reverse cordlist for b in concat(firstmon, nBasis)]
      antc := primitivePart antc
      result := concat(antc, result)
      ltresult := concat(antc-reductum antc, ltresult)
    else
      pivots(i) := j
      setRow_(linmat, i, lm)
      i := i + 1
      nBasis := cons(firstmon, nBasis)
  end if
end while

---- Compute the univariate polynomial for x
---- oldBasis is a total degree Groebner basis
minPol(oldBasis: List HDPoly, x: OV) : HDPoly ==
  algBasis := computeBasis oldBasis
  minPol(oldBasis, algBasis, x)

---- Compute the univariate polynomial for x
---- oldBasis is total Groebner, algBasis is the basis as algebra
minPol(oldBasis: List HDPoly, algBasis: List HDPoly, x: OV) : HDPoly ==
nvp: HDPoly := x::HDPoly
f := 1$HDPoly
omult : F := 1
ndim := (#algBasis) :: PositiveInteger
ndim1 := ndim + 1
lm := VF
linmat := zero(ndim, 2*ndim + 1)
linmat(1, 1) := 1$F
linmat(1, ndim1) := 1
pivots := Vector Integer := new(ndim, 0)
pivots(1) := 1
for i in 2..ndim1 repeat
  recf := rRedPol(f*nvp, oldBasis)
omult := recf.mult * omult
f := recf.poly
cc := gcd(content f, omult)
f := (f exquo cc) :: HDPoly
omult := (omult exquo cc) :: F
veccoef := VF := coord(f, algBasis)
lm := zero(2*ndim + 1)
for j in 1..ndim repeat lm(j) := veccoef(j)
lm(ndim + i) := omult
lm := reduceRow(linmat, lm, i-1, pivots)
j := 1
while lm(j) = 0 and j < ndim1 repeat j := j + 1
if j = ndim1 then return
  g := HDPoly := 0
  for k in ndim1..2*ndim1 + 1 repeat
    g := g + lm(k) * nvp**((k - ndim1) :: NNI)
  primitivePart g
  pivots(i) := j
  setRow_!(linmat, i, lm)

----- transform a DPoly in a HDPoly -----
transform(dpol : DPoly) : HDPoly ==
dpol = 0 => 0$HDPoly
monomial(leadingCoefficient dpol,
directProduct(degree(dpol) :: VV)$HDP)$HDPoly +
  transform(reductum dpol)

----- compute the basis for the vector space determined by b -----
computeBasis(B : List HDPoly) : List HDPoly ==
mB := List HDPoly := [monomial(1$F, degree f)$HDPoly for f in B]
result := List HDPoly := [1$HDPoly]
for var in lvar repeat
  part := intcompBasis(var, result, mB)
  result := concat(result, part)
result

----- internal function for computeBasis -----
intcompBasis(x : OV, lr : List HDPoly, mB : List HDPoly) : List HDPoly ==
lr=\[] => lr
\text{part:List HDPoly :=[]}\]
\text{for } f \text{ in } lr \text{ repeat }
g:=x::\text{HDPoly} * f
\text{if redPo}(g,\text{mB}).\text{poly}^{=}0 \text{ then part:=concat(g,part)}
\text{concat}(\text{part},\text{intcompBasis}(x,\text{part},\text{mB}))

\text{----- coordinate of } f \text{ with respect to the basis } B \text{ -----}
\text{----- } f \text{ is a reduced polynomial -----}
\text{coord}(f:\text{HDPoly},B:\text{List HDPoly}) : \text{VF} ==
\text{ndim} := \#B
\text{vv:VF}:=\text{new}(\text{ndim},0$F)$VF$\text{VF}
\text{while } f^{=}0 \text{ repeat }
\text{rf} := \text{reductum } f
\text{lf} := f-rf
\text{lcf} := \text{leadingCoefficient } f
i:Z:=\text{position}(\text{monomial}(1$F$,\text{degree } lf),B)
\text{vv}.i:=lcf
f := rf
\text{vv}

\text{----- reconstruct the polynomial from its coordinate -----}
\text{anticoord}(\text{vv:List F},mf:DPoly,B:List DPoly) : DPoly ==
\text{for } f \text{ in } B \text{ for } c \text{ in } \text{vv} \text{ repeat } (mf:=mf-c*f)
mf

\text{----- choose the next monom -----}
\text{chooseMon}(mf:DPoly,nB:List DPoly) : DPoly ==
nB = \[] => ((lvar.last)::DPoly)*mf
\text{for } x \text{ in reverse } lvar \text{ repeat }
\text{xx}:=x ::DPoly
\text{mf}:=\text{xx*mf}
\text{if redPo}(mf,nB).\text{poly}^{=}0 \text{ then return } mf
\text{dx} := \text{degree}(mf,x)
\text{mf} := (mf \text{exquo} (xx \text{**} dx))::DPoly
mf

\text{----- put } B \text{ in general position, } B \text{ is Groebner -----}
\text{linGenPos}(B : \text{List HDPoly}) : \text{LVals} ==
\text{result:List DPoly :=[]}
\text{tresult:List DPoly :=[]}
\text{vBasis}:= \text{computeBasis } B
\text{nBasis:List DPoly} :=[1$DPoly]
\text{ndim}:=\#\text{vBasis} : \text{PositiveInteger}
\text{ndim1}:\text{NNI}:=\text{ndim+1}
lm:VF
\text{linmat:MF}:=\text{zero}(\text{ndim},2*\text{ndim}+1)
\text{linmat(1,1)}:=1$F$
\text{linmat(1,\text{ndim}1)}:=1
\text{pivots:Vector Integer} := \text{new}(\text{ndim},0)
pivots(1) := 1
i:NNI:=2
rval:List Z :=[]
for ii in 1.(#lvar-1) repeat
  c:Z:=0
  while c=0 repeat c:=random()$Z rem 11
  rval:=concat(c,rval)
  nval:DPoly := (last.lvar)::DPoly -
    (+*[r*(vv)::DPoly for r in rval for vv in lvar])
firstmon:DPoly:=1$DPoly
ofirstmon:DPoly:=1$DPoly
orecfmon:Record(poly:HDPoly, mult:F) := [1,1]
lx:= lvar.last
while (firstmon:=choosemon(firstmon,ltresult))\=$1 repeat
  if (v:=firstmon exquo ofirstmon) case "failed" then
    recfmon:=rRedPol(transform(eval(firstmon,lx,nval)),B)
  else
    recfmon:=rRedPol(transform(eval(v,lx,nval))*orecfmon.poly,B)
    recfmon.poly := (recfmon.poly exquo cc)::HDpoly
    recfmon.mult := (recfmon.mult exquo cc)::F
  veccoef:VF:=coord(recfmon.poly,vBasis)
ofirstmon:=firstmon
orecfmon := recfmon
lm:=zero(2*ndim+1)
for j in 1..ndim repeat lm(j):=veccoef(j)
lm(2*ndim+i):=recfmon.mult
lm := reduceRow(linmat, lm, i-1, pivots)
j:=1
while lm(j) = 0 and j<ndim1 repeat j:=j+1
if j=ndim1 then
  cordlist:List F:=[lm(j) for j in ndim1..ndim1+#nBasis]
  antc:=+/[c*b for c in reverse cordlist
    for b in concat(firstmon,nBasis)]
  result:=concat(primitivePart antc,result)
  ltresult:=concat(antc-reductum antc,ltresult)
else
  pivots(i) := j
  setRow_!(linmat,i,lm)
i:=i+1
nBasis:=concat(firstmon,nBasis)
[result,rval]$LVals

----- given a basis of a zero-dimensional ideal,
----- performs a random change of coordinates
----- computes a Groebner basis for the lex ordering
groebgen(L:List DPoly) : cLVars ==
xn:=lvar.last
val := xn::DPoly
package LOP LinesOpPack

— LinesOpPack.input —

)set break resume
)sys rm -f LinesOpPack.output
)spool LinesOpPack.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinesOpPack
--R
--R LinesOpPack(K: Field) is a package constructor
--R Abbreviation for LinesOpPack is LOP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LOP
--R
--R----------------------------------- Operations -----------------------------------
--R quotVecSpaceBasis : (List(List(K)),List(List(K))) -> List(List(K))
--R reduceLineOverLine : (List(K),List(K),K) -> List(K)
--R reduceRow : List(List(K)) -> List(List(K))
--R reduceRowOnList : (List(K),List(List(K))) -> List(List(K))
--R rowEchWoZeroLines : Matrix(K) -> Matrix(K)

nvar1:NNI:=(#lvar-1):NNI
ll: List Z :=[random()$Z rem 11 for i in 1..nvar1]
val:=val+ +/[ll.i*(lvar.i)::DPoly for i in 1..nvar1]
LL:=[elt(univariate(f,xn),val) for f in L]
LL:= groebner(LL)
[LL,ll]$cLVars

— LGROBP.dotabb —

"LGROBP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LGROBP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LGROBP" -> "DIRPCAT"
"LGROBP" -> "PFECAT"
A package that exports several linear algebra operations over lines of matrices. Part of the PAFF package.

See Also:
- )show LinesOpPack

---

LOP

**Exports:**
- quotVecSpaceBasis
- reduceLineOverLine
- reduceRow
- reduceRowOnList
- rowEchWoZeroLines
- rowEchWoZeroLinesWOVectorise

---

)abbrev package LOP LinesOpPack
++ Authors: G. Hache
++ Date Created: 21 sept 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that exports several linear algebra operations over lines
++ of matrices. Part of the PAFF package.

LinesOpPack(K):P==T where

K:Field

P== with
    rowEchWoZeroLinesWoVectorise: Matrix(K) -> Matrix(K)
    rowEchWoZeroLines: Matrix(K) -> Matrix(K)
    reduceRow: List(List(K)) -> List(List(K))
        ++ reduceRow: if the input is considered as a matrix, the output would
        ++ be the row reduction matrix. It’s almost the rowEchelon form
        ++ except that no permuation of lines is performed.
    quotVecSpaceBasis: (List(List(K)),List(List(K))) -> List(List(K))
        ++ quotVecSpaceBasis(b1,b2) returns a basis of V1/V2 where
        ++ V1 and V2 are vector space with basis b1 and b2 resp. and
        ++ V2 is suppose to be include in V1; Note that if
        ++ it is not the case then it returns the basis of V1/W
        ++ where W = intersection of V1 and V2
    reduceLineOverLine: (List(K),List(K),K) -> List(K)
        ++ reduceLineOverLine(v1,v2,a) returns v1-a*v1 where
        ++ v1 and v2 are considered as vector space.
    reduceRowOnList: (List(K),List(List(K))) -> List(List(K))
        ++ reduceRowOnList(v,lvec) applies a row reduction on each of the
        ++ element of lv using v according to a pivot in v which is set to
        ++ be the first non nul element in v.

T==> add
    localRowEchelon: Matrix(K) -> Matrix(K)
    localRowEchelon(m)==
        "(K has PseudoAlgebraicClosureOfPerfectFieldCategory ) => rowEchelon m"
            llm:List(List(K)):= listOfLists m
            l:= first llm
            maxT:= maxTower l
            lv := [vectorise(a,maxT)$K for a in l]
            subMat1 := transpose matrix [entries(v) for v in lv]
            mat1:= subMat1
            for l in rest llm repeat
                maxT:= maxTower l
                lv := [vectorise(a,maxT)$K for a in l]
                subMat1 := transpose matrix [entries(v) for v in lv]
                mat1:=vertConcat(mat1,subMat1)
                rowEchelon mat1

    rowEchWoZeroLines(m)==
        mm:=localRowEchelon m
        ll:=listOfLists mm
n := # first ll
lZero := new(n pretend NonNegativeInteger, 0)$List(K)
llll := [ l for l in ll | ^(lZero = l) ]
empty?(llll) => matrix [lZero]
matrix llll

rowEchWoZeroLinesWOVectorise(m) ==
mm := rowEchelon m
ll := listOfLists mm
n := # first ll
lZero := new(n pretend NonNegativeInteger, 0)$List(K)
llll := [ l for l in ll | ^(lZero = l) ]
empty?(llll) => matrix [lZero]
matrix llll

quotVecSpaceBasis(l2, l1) ==
redBasis := reduceRow(concat(l1, l2))
tempRes := rest(redBasis, #11)
allZero := new(#l1.1, 0$K)
[l for l in tempRes | ^(l = allZero)]

reduceRowOnList(line, listOfLine) ==
frsNonNul := position("zero?"(#1), line)
^((frsNonNul > 0) => listOfLine)
a := line.frsNonNul
inva := inv a
newLine := [inva*c for c in line]
[reduceLineOverLine(newLine, l, l.frsNonNul) for l in listOfLine]

reduceLineOverLine(l1, l2, b) ==
[c2 - b*c1 for c2 in l2 for c1 in l1]

reduceRow(m: List(List(K))) ==
n := #m
mcopy := copy m
newBottom := List(List(K))
for i in 1..(n-1) repeat
newBottom := reduceRowOnList(mcopy.i, [mcopy.j for j in (i+1)..n])
mcopy := concat([mcopy.k for k in 1..i] :: List(List(K)), newBottom)
mcopy

———

— LOP.dotabb ——

"LOP" [color="#FF4488", href="bookvol10.4.pdf#nameddest=LOP"]
"IVECTOR" [color="#88FF44", href="bookvol10.3.pdf#nameddest=IVECTOR"]
"LOP" -> "IVECTOR"
package LF LiouvillianFunction

---

LiouvillianFunction examples

This package provides liouvillian functions over an integral domain.

See Also:
- )show LiouvillianFunction

---
LiouvillianFunction (LF)

Exports:
belong?  Ci  dilog  Ei  erf
integral  li  integral  operator  Si

— package LF LiouvillianFunction —

)abbrev package LF LiouvillianFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 August 1994
++ Description:
++ This package provides liouvillian functions over an integral domain.

LiouvillianFunction(R, F):Exports == Implementation where
R:Join(OrderedSet, IntegralDomain)
F:Join(FunctionSpace R,RadicalCategory,TranscendentalFunctionCategory)

OP ==> BasicOperator
PR ==> Polynomial R
K ==> Kernel F
SE ==> Symbol
O ==> OutputForm
INP ==> InputForm
INV ==> error "Invalid argument"

SPECIALDIFF ==> "%specialDiff"
SPECIALDISP ==> "%specialDisp"
SPECIALINPUT ==> "%specialInput"
SPECIALEQUAL ==> "%specialEqual"

Exports == with
  belong? : OP -> Boolean
  ++ belong?(op) checks if op is Liouvillian
  operator: OP -> OP
  ++ operator(op) returns the Liouvillian operator based on op
Ei : F -> F
++ Ei(f) denotes the exponential integral
Si : F -> F
++ Si(f) denotes the sine integral
Ci : F -> F
++ Ci(f) denotes the cosine integral
li : F -> F
++ li(f) denotes the logarithmic integral
erf : F -> F
++ erf(f) denotes the error function
dilog : F -> F
++ dilog(f) denotes the dilogarithm
fresnelS : F -> F
++ fresnelS(f) denotes the Fresnel integral S
fresnelC : F -> F
++ fresnelC(f) denotes the Fresnel integral C
integral: (F, SE) -> F
++ integral(f,x) indefinite integral of f with respect to x.
integral: (F, SegmentBinding F) -> F
++ integral(f,x = a..b) denotes the definite integral of f with
++ respect to x from \spad{a} to b.

Implementation ==> add
iei : F -> F
isi : F -> F
ici : F -> F
ierf : F -> F
ili : F -> F
ili2 : F -> F
iint : List F -> F
eqint : (K,K) -> Boolean
dvint : (List F, SE) -> F
dvdint : (List F, SE) -> F
ddint : List F -> O
integrand : List F -> F
dummy := new()$SE :: F

opint := operator("integral"::Symbol)$CommonOperators
opdint := operator("defint"::Symbol)$CommonOperators
opei := operator("Ei"::Symbol)$CommonOperators
opl := operator("li"::Symbol)$CommonOperators
opsi := operator("Si"::Symbol)$CommonOperators
opc := operator("Ci"::Symbol)$CommonOperators
opi2 := operator("dilog"::Symbol)$CommonOperators
operf := operator("erf"::Symbol)$CommonOperators
opfis := operator("fresnelS"::Symbol)$CommonOperators
opfic := operator("fresnelC"::Symbol)$CommonOperators

Si x == opsi x
Ci x == opci x
Ei x == opei x
erf x == operf x
li x == opli x
dilog x == opli2 x
fresnelS x == opfis x
fresnelC x == opfic x

belong? op == has?(op, "prim")
isì x == kernel(opsi, x)
ici x == kernel(opci, x)
ierf x == (zero? x => 0; kernel(operf, x))
--  ili2 x == (one? x => INV; kernel(opli2, x))

ifis (x:F):F == (zero? x => 0; kernel(opfis, x))
ific (x:F):F == (zero? x => 0; kernel(opfic, x))

integrand l == eval(first l, retract(second l)@K, third l)
integral(f:F, x:SE) == opint [eval(f, k:=kernel(x)$K, dummy), dummy, k::F]

iint l ==
  zero? first l => 0
  kernel(opint, l)

dint l ==
  int(integrand(l)::O * hconcat("d"::SE::O, third(l)::O),
  third(rest l)::O, third(rest rest l)::O)
eqint(k1, k2) ==
  a1:=argument k1
  a2:=argument k2
  res:=operator k1 = operator k2
  if not res then return res
  res:= a1 = a2
  if res then return res
  res:= (a1.3 = a2.3) and (subst(a1.1,[retract(a1.2)@K],[a2.2]) = a2.1)

dvint(l, x) ==
  k := retract(second l)@K
  differentiate(third l, x) * integrand l
  + opint [differentiate(first l, x), second l, third l]

dvdint(l, x) ==
  x = retract(y := third l)@SE => 0
  k := retract(d := second l)@K
  differentiate(h := third rest rest l, x) * eval(f := first l, k, h)
  - differentiate(g := third rest l, x) * eval(f, k, g)
  + opdint [differentiate(f, x), d, y, g, h]

integral(f:F, s: SegmentBinding F) ==
\begin{verbatim}
x := kernel(variable s)$K
opdint [eval(f,x,dummy), dummy, x::F, lo segment s, hi segment s]

ili x ==
x = 1 \rightarrow INV
is?(x, "exp"::Symbol) \rightarrow Ei first argument(retract(x)@K)
kernel(opli, x)

iei x ==
x = 0 \rightarrow INV
is?(x, "log"::Symbol) \rightarrow li first argument(retract(x)@K)
kernel(opei, x)

operator op ==
is?(op, "integral"::Symbol) \rightarrow opint
is?(op, "%defint"::Symbol) \rightarrow opdint
is?(op, "Ei"::Symbol) \rightarrow opei
is?(op, "Si"::Symbol) \rightarrow opsi
is?(op, "Ci"::Symbol) \rightarrow opci
is?(op, "li"::Symbol) \rightarrow opli
is?(op, "erf"::Symbol) \rightarrow operf
is?(op, "dilog"::Symbol) \rightarrow opli2
is?(op, "fresnelC"::Symbol) \rightarrow opfis
is?(op, "fresnelS"::Symbol) \rightarrow opfic
error "Not a Liouvillian operator"
evaluate(opei, iei)$BasicOperatorFunctions1(F)
evaluate(opli, ili)
evaluate(opsi, isi)
evaluate(opci, ici)
evaluate(operf, ierf)
evaluate(opli2, ili2)
evaluate(opfis, ifis)
evaluate(opfic, ific)
evaluate(opint, iint)
derivative(opsi, (z1:F):F +-> sin(z1) / z1)
derivative(opci, (z1:F):F +-> cos(z1) / z1)
derivative(opei, (z1:F):F +-> exp(z1) / z1)
derivative(opli, (z1:F):F +-> inv log(z1))
derivative(operf, (z1:F):F +-> 2 * exp(-((z1**2)) / sqrt(pi()))
derivative(opli2, (z1:F):F +-> log(z1) / (1 - z1))
derivative(opfis, (z1:F):F +-> log(z1) / (1 - z1))
derivative(opfic, (z1:F):F +-> sin(z1**2))
derivative(opfis, (z1:F):F +-> log(z1) / (1 - z1))
setProperty(opint,SPECIALEQUAL,eqint@((K,K) -> Boolean) pretend None)
setProperty(opint,SPECIALDIFF,dvint@((List F,SE) -> F) pretend None)
setProperty(opdint,SPECIALDIFF,dvint@((List F,SE) -> F) pretend None)
setProperty(opdint,SPECIALDISP, ddint@(List F -> O) pretend None)
if R has ConvertibleTo INP then
  inint : List F -> INP
\end{verbatim}
\textbf{CHAPTER 13. CHAPTER L}

\begin{verbatim}

indint: List F -> INP
pint : List INP -> INP

pint l == convert concat(convert("integral":SE)@INP, l)
inint l ==
  r2:= convert(
    [convert("":SE)@INP,
     convert(third l)@INP,
     convert("Symbol":SE)@INP]@List INP)@INP
  pint [convert(integrand l)@INP, r2]

indint l ==
pint [convert(integrand l)@INP,
  convert concat(convert("=":SE)@INP,
    [convert(third l)@INP,
     convert concat(convert("SEGMENT":SE)@INP,
       [convert(third rest l)@INP,
        convert(third rest rest l)@INP])])

setProperty(opint, SPECIALINPUT, inint@(List F -> INP) pretend None)
setProperty(opdint, SPECIALINPUT, indint@(List F -> INP) pretend None)

——

— LF.dotabb —

"LF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"LF" -> "FS"

——

package LIST2 ListFunctions2

— ListFunctions2.input —

)set break resume
)sys rm -f ListFunctions2.output
)spool ListFunctions2.output
)set message test on
)set message auto off
)clear all
\end{verbatim}
ListFunctions2 implements utility functions that operate on two kinds of lists, each with a possibly different type of element.

See Also:
- )show ListFunctions2

---

ListFunctions2 (LIST2)

Exports:
- map
- reduce
- scan

---

)abbrev package LIST2 ListFunctions2
++ Description:
**ListFunctions2** implements utility functions that operate on two kinds of lists, each with a possibly different type of element.

**ListFunctions2**(*A*: Type, *B*: Type): public == private where

*LA* ==&gt; List *A*

*LB* ==&gt; List *B*

*O2* ==&gt; FiniteLinearAggregateFunctions2(*A*, *LA*, *B*, *LB*)

public ==&gt; with

scan: ((*A*, *B*) &rarr; *B*, *LA*, *B*) &rarr; *LB*

++ scan(*fn*, *u*, *ident*) successively uses the binary function
++ \spad{fn} to reduce more and more of list \spad{u}.
++ \spad{ident} is returned if the \spad{u} is empty.
++ The result is a list of the reductions at each step. See
++ \spadfun{reduce} for more information. Examples:
++ \spad{scan(*fn*, [{1, 2}], 0) = \{fn(2, fn(1, 0)), fn(1, 0)\}} and
++ \spad{scan(*, [{2, 3}], 1) = \{2 * 1, 3 * (2 * 1)\}}.

reduce: ((*A*, *B*) &rarr; *B*, *LA*, *B*) &rarr; *B*

++ reduce(*fn*, *u*, *ident*) successively uses the binary function
++ \spad{fn} on the elements of list \spad{u} and the result
++ of previous applications. \spad{ident} is returned if the
++ \spad{u} is empty. Note the order of application in
++ the following examples:
++ \spad{reduce(*fn*, [{1, 2, 3}], 0) = fn(3, fn(2, fn(1, 0)))} and
++ \spad{reduce(*, [{2, 3}], 1) = 3 * (2 * 1)}.

map: (*A* &rarr; *B*, *LA*) &rarr; *LB*

++ map(*fn*, *u*) applies \spad{fn} to each element of
++ list \spad{u} and returns a new list with the results.
++ For example \spad{map(square, [{1, 2, 3}]) = \{1, 4, 9\}}.

private ==&gt; add

map(*f*, *l*) == map(*f*, *l*)$O2

scan(*f*, *l*, *b*) == scan(*f*, *l*, *b*)$O2

reduce(*f*, *l*, *b*) == reduce(*f*, *l*, *b*)$O2

---

LIST2.dotabb

"LIST2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIST2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"LIST2" -> "FLAGG"

---
package ListFunctions3

--- ListFunctions3.input ---

)set break resume
)sys rm -f ListFunctions3.output
)spool ListFunctions3.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ListFunctions3
--E 1

)spool
)lisp (bye)

---

--- ListFunctions3.help ---

====================================================================
ListFunctions3 examples
====================================================================

ListFunctions3 implements utility functions that operate on three kinds of lists, each with a possibly different type of element.

See Also:
  o )show ListFunctions3

---
ListFunctions3 (LIST3)

Exports:
map

--- package LIST3 ListFunctions3 ---

)abbrev package LIST3 ListFunctions3
++ Description:
++ \texttt{ListFunctions3} implements utility functions that
++ operate on three kinds of lists, each with a possibly different
++ type of element.

ListFunctions3(A:Type, B:Type, C:Type): public == private where
LA ==> List A
LB ==> List B
LC ==> List C

class public == with
map: ( (A,B) -> C, LA, LB ) -> LC
++ map(fn,list1, u2) applies the binary function \texttt{fn}
++ to corresponding elements of lists \texttt{spad{u1}} and \texttt{spad{u2}}
++ and returns a list of the results (in the same order). Thus
++ \texttt{spad{map}(/,[1,2,3],[4,5,6]) = [1/4,2/4,1/2]}. The computation
++ terminates when the end of either list is reached. That is,
++ the length of the result list is equal to the minimum of the
++ lengths of \texttt{spad{u1}} and \texttt{spad{u2}}.

private == add
map(fn : (A,B) -> C, la : LA, lb : LB): LC ==
empty?(la) or empty?(lb) => empty()$LC
concat(fn(first la, first lb), map(fn, rest la, rest lb))
package LIST2MAP ListToMap

-- ListToMap.input --

)set break resume
)sys rm -f ListToMap.output
)spool ListToMap.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ListToMap
--E 1

)spool
)lisp (bye)

-- ListToMap.help --

====================================================================
ListToMap examples
====================================================================

ListToMap allows mappings to be described by a pair of lists of equal lengths. The image of an element x, which appears in position n in the first list, is then the n-th element of the second list. A default value or default function can be specified to be used when x does not appear in the first list. In the absence of defaults, an error will occur in that case.

See Also:
o )show ListToMap
ListToMap (LIST2MAP)

Exports:
match

— package LIST2MAP ListToMap —

)abbrev package LIST2MAP ListToMap
++ Author: Manuel Bronstein
++ Date Created: 22 Mar 1988
++ Change History: 11 Oct 1989
++ Description:
++ \spadtype{ListToMap} allows mappings to be described by a pair of
++ lists of equal lengths. The image of an element \spad{x},
++ which appears in position \spad{n} in the first list, is then
++ the \spad{n}th element of the second list. A default value or
++ default function can be specified to be used when \spad{x}
++ does not appear in the first list. In the absence of defaults,
++ an error will occur in that case.

ListToMap(A:SetCategory, B:Type):Exports == Implementation where
LA ==> List A
LB ==> List B
AB ==> (A -> B)

Exports == with
match: (LA, LB) -> AB
++ match(la, lb) creates a map with no default source or target values
++ defined by lists la and lb of equal length.
++ The target of a source value \spad{x} in la is the
++ value y with the same index lb.
++ Error: if la and lb are not of equal length.
++ Note that when this map is applied, an error occurs when
++ applied to a value missing from la.
match: (LA, LB, A) -> B
  ++ match(la, lb, a) creates a map
  ++ defined by lists la and lb of equal length, where \spad{a} is used
  ++ as the default source value if the given one is not in \spad{la}.
  ++ The target of a source value \spad{x} in la is the
  ++ value y with the same index lb.
  ++ Error: if la and lb are not of equal length.
match: (LA, LB, B) -> AB
  ++ match(la, lb, b) creates a map
  ++ defined by lists la and lb of equal length, where \spad{b} is used
  ++ as the default target value if the given function argument is
  ++ not in \spad{la}.
  ++ The target of a source value \spad{x} in la is the
  ++ value y with the same index lb.
  ++ Error: if la and lb are not of equal length.
match: (LA, LB, A, B) -> B
  ++ match(la, lb, a, b) creates a map
  ++ defined by lists la and lb of equal length.
  ++ and applies this map to a.
  ++ The target of a source value \spad{x} in la is the
  ++ value y with the same index lb.
  ++ Argument b is the default target value if a is not in la.
  ++ Error: if la and lb are not of equal length.
match: (LA, LB, AB) -> AB
  ++ match(la, lb, f) creates a map
  ++ defined by lists la and lb of equal length.
  ++ The target of a source value \spad{x} in la is the
  ++ value y with the same index lb.
  ++ Argument \spad{f} is used as the
  ++ function to call when the given function argument is not in
  ++ \spad{la}.
  ++ The value returned is \spad{f} applied to that argument.
match: (LA, LB, A, AB) -> B
  ++ match(la, lb, a, f) creates a map
  ++ defined by lists la and lb of equal length.
  ++ and applies this map to a.
  ++ The target of a source value \spad{x} in la is the
  ++ value y with the same index lb.
  ++ Argument \spad{f} is a default function to call if a is not in la.
  ++ The value returned is then obtained by applying f to argument a.

Implementation ==> add
match(la, lb)          == (z1:A):B +-> match(la, lb, z1)
match(la:LA, lb:LB, a:A) == lb.position(a, la)
match(la:LA, lb:LB, b:B) == (z1:A):B +-> match(la, lb, z1, b)
match(la:LA, lb:LB, f:AB) == (z1:A):B +-> match(la, lb, z1, f)
match(la:LA, lb:LB, a:A, b:B) ==
(p := position(a, la)) < minIndex(la) => b
lb.p

match(la:LA, lb:LB, a:A, f:AB) ==
  (p := position(a, la)) < minIndex(la) => f a
lb.p

package LPARSPT LocalParametrizationOfSimplePoint-Package

-- LocalParametrizationOfSimplePointPackage.input --

)set break resume
/sys rm -f LocalParametrizationOfSimplePointPackage.output
/spool LocalParametrizationOfSimplePointPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show LocalParametrizationOfSimplePointPackage
--R
--R LocalParametrizationOfSimplePointPackage(K: Field, symb: List(Symbol),PolyRing: PolynomialCategory,ProjPt: ProjectiveSpaceCategory(K),PCS: LocalPowerSeriesCategory(K),Plc: PlacesCategory(K,PCS)) is a package constructor
--R Abbreviation for LocalParametrizationOfSimplePointPackage is LPARSPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LPARSPT
--R
--R------------------------------- Operations --------------------------------
--R pointDominateBy : Plc -> ProjPt  printInfo : Boolean -> Boolean
--R printInfo : () -> Boolean
--R localize : (PolyRing,ProjPt,PolyRing,Integer) -> Record(fnc: PolyRing,crv: PolyRing,chart: List(Integer))
--R pointToPlace : (ProjPt,PolyRing) -> Plc
LocalParametrizationOfSimplePointPackage (LPARSPT)

Exports:
- pointDominateBy
- printInfo
- localParamOfSimplePt
- localize
- pointToPlace

See Also:
- )show LocalParametrizationOfSimplePointPackage

---

package LPARSPT LocalParametrizationOfSimplePointPackage

)abbrev package LPARSPT LocalParametrizationOfSimplePointPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This package is part of the PAFF package
LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc):_
Exports == Implementation where
K:Field
  symb: List(Symbol)
E : DirectProductCategory(#symb,NonNegativeInteger)
OV ==> OrderedVariableList(symb)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
PackPoly ==> PackageForPoly(K,PolyRing,E,#symb)
UP ==> SparseUnivariatePolynomial(K)
UPUP ==> SparseUnivariatePolynomial(UP)
Exports == with

printInfo: Boolean -> Boolean
  ++ printInfo(b) set a flag such that when true (b <- true) prints
  ++ some information during some critical computation.
printInfo: () -> Boolean
  ++ returns the value of the \spad{printInfo} flag.

pointToPlace: (ProjPt,PolyRing) -> Plc
  ++ pointToPlace(pt,pol) takes for input a simple point pt on the curve
  ++ defined by pol and set the local parametrization of the point.

localParamOfSimplePt: (ProjPt,PolyRing,Integer) -> List PCS
  ++ localParamOfSimplePt(pt,pol,n) computes the local parametrization
  ++ of the simple point pt on the curve defined by pol. This local
  ++ parametrization is done according to the standard open affine
  ++ plane set by n

pointDominateBy : Plc -> ProjPt
  ++ pointDominateBy(pl) returns the projective point dominated
  ++ by the place pl.

localize: (PolyRing,ProjPt,PolyRing,Integer) -> _
  Record(fnc:PolyRing,crv:PolyRing,chart:List(Integer))
  ++ localize(f,pt,crv,n) returns a record containing the polynomials f
  ++ and crv translate to the origin with respect to pt. The last
++ element of the records, consisting of three integers contains
++ information about the local parameter that will be used
++ (either x or y): the first integer correspond to the variable
++ that will be used as a local parameter.

Implementation ==> add

import PCS
import PolyRing
import PPFC1
import PackPoly

valuationAndMore: (UPUP,UPUP) -> _
   Record(ord:Integer,value:K,fnc:UPUP,crv:UPUP)

localize2: (PolyRing,ProjPt,PolyRing,Integer) -> _
   Record(fnc2:UPUP,crv2:UPUP)

coerceToUPUP: (PolyRing,List Integer) -> UPUP

paramAtOrigin: (UPUP,UPUP,Integer) -> PCS

strictTransform: (UPUP,NNI) -> UPUP

translate: (UPUP,K) -> UPUP

constant: UPUP -> K

intCoord: UPUP -> K

localMultiplicity: UPUP -> NNI

mapDegree: (NNI,NNI,NNI) -> NNI

listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]

listMonoPols:List(PolyRing):=listVariable()

pointDominateBy(pl)==
   lpl:List PCS:=localParam(pl)
   empty? lpl => _
   error "LPARSPT:pointDominateBy::parametrization of point not done yet"
   lK:List K:=[ findCoef(s,0) for s in lpl]
   projectivePoint(lK)

localParamOfSimplePt(pt,curve,nV)==
   mult:NNI:=multiplicity(curve,pt,nV)
   "one?"(mult) => _
   error "The point is not simple or is not on the curve !"
   lcl:=[localize2(var,pt,curve,nV) for var in listMonoPols]
pointToPlace(pt,curve)==
    -- define the chart for strictTransform (of simple point)
    nV:Integer:=lastNonNull pt
    pth:=homogenize(pt,nV)
    chart:List Integer:=[0,0,nV]
    mult:NNI:=multiplicity(curve,pth,nV)
    "one?(mult) =>
        error "The point is not simple or is not on the curve"
    -- create a place from the simple point. This is done by giving
    -- a name to the place: in this case it is the coordinate of
    -- the projective point.
    lpth:List K:= pth :: List(K)
    plc:Plc:=create(lpth)$Plc
    "empty?(localParam(plc)) => plc
    lcl:=[localize2(var,pt,curve,nV) for var in listMonoPols]
    lPar:=[paramAtOrigin(1.fnc2,1.crv2,0) for l in lcl]
    setParam!(plc,lPar)
    dd:=degree pth
    setDegree!(plc,dd)
    plc

localVarForPrintInfo:Boolean:=false()$Boolean

printInfo()=localVarForPrintInfo
printInfo(flag)=localVarForPrintInfo:=flag

mapDegree(n,mx,m)==
    dd:=(n+mx-m)
    dd < 0 => _
    error "LPARSPT:mapDegree called by PARAMP:strictTransform failed"
    dd pretend NNI

strictTransform(pol,m)==
    zero?(pol) => 0
    tc:=leadingCoefficient pol
    tk:= degree pol
    newTc:= mapExponents(mapDegree(#1,tk,m),tc)
    monomial(newTc,tk)$UPUP + strictTransform(reductum pol,m)

Y == monomial(1,1)$UPUP

trY: (K,NonNegativeInteger) -> UPUP
trY(a,n)== (monomial(monomial(a,0)$UP,0)$UPUP + Y)**n

translate(pol,a)==
    zero?(pol) => 0
    tc:=leadingCoefficient pol
tk := degree pol
tkY(a, tk) * tc + translate(reductum pol, a)

constant(pol) == coefficient(coefficient(pol, 0)$UPUP, 0)$UP

intCoord(pol) ==
  coefY := coefficient(coefficient(pol, 1)$UPUP, 0)$UP
  cnst := constant(pol)
  -cnst * inv coefY

localMultiplicity(pol) ==
  zero?(pol) => error "Cannot compute the multiplicity for 0"
  redPol := reductum pol
  tc := leadingCoefficient pol
  tk := degree pol
  m := tk + minimumDegree(tc)$UP
  zero?(redPol) => m
  min(m, localMultiplicity(redPol))

coerceToUPUP(pol, chart) ==
  zero?(pol) => 0
  lExp := parts degree pol
  lCoef := leadingCoefficient pol
  expX := lExp(chart.1)
  expY := lExp(chart.2)
  monomial(monomial(lCoef, expX)$UP, expY)$UPUP + _
  coerceToUPUP(reductum(pol), chart)

-- testing this function. See paramPack for original version.
valuationAndMore(f: UPUP, curve: UPUP) ==
  -- this function evaluate the function f at the origin
  -- which must be a simple point on the curve define by "curve"
  val := constant(f)
  "zero?(val) => [0, val, f, curve]
  sTrCurve := strictTransform(curve, 1)
  slp := intCoord sTrCurve
  multPtf := Integer := localMultiplicity(f) pretend Integer
  sTrFnc := strictTransform(f, multPtf pretend NNI)
  newCurve := translate(sTrCurve, slp)
  f2 := translate(sTrFnc, slp)
  val := constant(f2)
  [multPtf, val, f2, newCurve]

paramAtOrigin(f: UPUP, curve: UPUP, ex: Integer) == delay
  -- this function must be
  -- called for parametrization a the origin
  u := f
  zero?(u) => 0
  tt := u exquo curve
  "(tt case "failed") => 0
firstTerm := valuationAndMore(u, curve)
od := firstTerm.ord
coef := firstTerm.value
newU := firstTerm.fnc - monomial(monomial(coef, 0)$UP, 0)$UPUP
newCurve := firstTerm.crv
series(od + ex, coef, paramAtOrigin(newU, newCurve, ex + od))

localize(f: PolyRing, pt: ProjPt, curve: PolyRing, nV: Integer) ==
curveT := translateToOrigin(curve, pt, nV)
ft := translateToOrigin(f, pt, nV)
fm := minimalForm(curveT)
zero?(d := totalDegree(fm)$PackPoly) => _
   error "the point is not on the curve"
"one?(d) => error "the point is singular"
subChart := [i for i in 1..#symb | ^((i = (nV pretend PI)))]
cf1 := degOneCoef(fm, (subChart.1) pretend PI)
cf2 := degOneCoef(fm, (subChart.2) pretend PI)
crt := List(Integer)
sc := List(Integer):= [(i pretend Integer) for i in subChart]
zero?(cf1) =>
   crt := concat(sc, nV)
   [ft, curveT, crt]
zero?(cf2) =>
   crt := concat(reverse(sc), nV)
   [ft, curveT, crt]
der1 := degree(curveT, listVar(subChart.1))
der2 := degree(curveT, listVar(subChart.2))
der1 > der2 =>
   crt := concat(sc, nV)
   [ft, curveT, crt]
crt := concat(reverse(sc), nV)
   [ft, curveT, crt]

localize2(f: PolyRing, pt: ProjPt, curve: PolyRing, nV: Integer) ==
recBlowUp := localize(f, pt, curve, nV)
f2 := coerceToUPUP(recBlowUp.fnc, recBlowUp.chart)
curve2 := coerceToUPUP(recBlowUp.crv, recBlowUp.chart)
[f2, curve2]
Chapter 14

Chapter M

package MKBCFUNC MakeBinaryCompiledFunction

— MakeBinaryCompiledFunction.input —

)set break resume
)sys rm -f MakeBinaryCompiledFunction.output
)spool MakeBinaryCompiledFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MakeBinaryCompiledFunction
--E 1

)spool
)lisp (bye)

— MakeBinaryCompiledFunction.help —

====================================================================
MakeBinaryCompiledFunction examples
====================================================================

Tools and transforms for making compiled functions from top-level expressions

See Also:

1453
MakeBinaryCompiledFunction (MKBCFUNC)

Exports:
  binaryFunction  compiledFunction

— package MKBCFUNC MakeBinaryCompiledFunction —

)abbrev package MKBCFUNC MakeBinaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools and transforms for making compiled functions from
++ top-level expressions

MakeBinaryCompiledFunction(S, D1, D2, I):Exports == Implementation where
  S: ConvertibleTo InputForm
  D1, D2, I: Type

SY  ==> Symbol
DI  ==> devaluate((D1, D2) -> I)$Lisp

Exports == with
  binaryFunction : SY -> ((D1, D2) -> I)
     ++ binaryFunction(s) is a local function
  compiledFunction: (S, SY, SY) -> ((D1, D2) -> I)
     ++ compiledFunction(expr,x,y) returns a function \spad{f: (D1, D2) -> I}
        ++ defined by \spad{f(x, y) == expr}.
     ++ Function f is compiled and directly
++ applicable to objects of type \spad{(D1, D2)}

Implementation ==> add
import MakeFunction(S)

func: (SY, D1, D2) -> I

func(name, x, y) == FUNCALL(name, x, y, NIL$Lisp)$Lisp
binaryFunction name == (d1:D1,d2:D2):I +-> func(name, d1, d2)

compiledFunction(e, x, y) ==
t := [devaluate(D1)$Lisp, devaluate(D2)$Lisp]$List(InputForm)
binaryFunction compile(function(e, declare DI, x, y), t)

---

— MKBCFUNC.dotabb —

"MKBCFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKBCFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKBCFUNC" -> "KONVERT"
"MKBCFUNC" -> "TYPE"

---

package MKFLCFN MakeFloatCompiledFunction

— MakeFloatCompiledFunction.input —

)set break resume
)sys rm -f MakeFloatCompiledFunction.output
)spool MakeFloatCompiledFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MakeFloatCompiledFunction
--E 1

)spool
)lisp (bye)
MakeFloatCompiledFunction examples
====================================================================
Tools for making compiled functions from top-level expressions
MakeFloatCompiledFunction transforms top-level objects into compiled
Lisp functions whose arguments are Lisp floats. This by-passes the
Axiom compiler and interpreter, thereby gaining several orders of
magnitude.

See Also:
  o )show MakeFloatCompiledFunction

MakeFloatCompiledFunction (MKFLCFN)

Exports:

  — package MKFLCFN MakeFloatCompiledFunction —

)abbrev package MKFLCFN MakeFloatCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 2 Mar 1990
++ Date Last Updated: 2 Dec 1996 (MCD)
++ Description:
++ Tools for making compiled functions from top-level expressions
++ MakeFloatCompiledFunction transforms top-level objects into
++ compiled Lisp functions whose arguments are Lisp floats.
++ This by-passes the Axiom compiler and interpreter,
++ thereby gaining several orders of magnitude.

MakeFloatCompiledFunction(S): Exports == Implementation where
S: ConvertibleTo InputForm

INF ==> InputForm
SF ==> DoubleFloat
DI1 ==> devaluate(SF -> SF)$Lisp
DI2 ==> devaluate((SF, SF) -> SF)$Lisp

Exports ==> with
makeFloatFunction: (S, Symbol) -> (SF -> SF)
++ makeFloatFunction(expr, x) returns a Lisp function
++ \spad{f: \axiomType{DoubleFloat} -> \axiomType{DoubleFloat}}
++ defined by \spad{f(x) == expr}.
++ Function f is compiled and directly
++ applicable to objects of type \axiomType{DoubleFloat}.
makeFloatFunction: (S, Symbol, Symbol) -> ((SF, SF) -> SF)
++ makeFloatFunction(expr, x, y) returns a Lisp function
++ \spad{f: ((\axiomType{DoubleFloat},
++ \axiomType{DoubleFloat}) -> \axiomType{DoubleFloat})}
++ defined by \spad{f(x, y) == expr}.
++ Function f is compiled and directly
++ applicable to objects of type \spad{((\axiomType{DoubleFloat},
++ \axiomType{DoubleFloat})}.

Implementation ==> add
import MakeUnaryCompiledFunction(S, SF, SF)
import MakeBinaryCompiledFunction(S, SF, SF, SF)

streq? : (INF, String) -> Boolean
streqlist?: (INF, List String) -> Boolean
gencode : (String, List INF) -> INF
mkLisp : INF -> Union(INF, "failed")
mkLispList: List INF -> Union(List INF, "failed")
mkDefun : (INF, List INF) -> INF
mkLispCall: INF -> INF
mkPretend : INF -> INF
mkCTOR : INF -> INF

lsf := convert([convert("DoubleFloat"::Symbol)@INF]$List(INF))@INF

streq?(s, st)  == s = convert(st::Symbol)@INF
gencode(s, 1)  == convert(concat(convert(s::Symbol)@INF, 1))@INF
streqlist?(s, 1) == member?(string symbol s, 1)

mkPretend form ==
  convert([convert("pretend"::Symbol), form, lsf]$List(INF))@INF
mkCTOR form ==
    convert([convert("C-TO-R"::Symbol), form]$List(INF))@INF

mkLispCall name ==
    convert([convert("$elt"::Symbol),
      convert("Lisp"::Symbol), name]$List(INF))@INF

mkDefun(s, lv) ==
    name := convert(new()$Symbol)@INF
    fun := convert([convert("DEFUN"::Symbol), name, convert lv,
      gencode("DECLARE",[gencode("FLOAT",lv)]),mkCTOR s]$List(INF))@INF
    EVAL(fun)$Lisp
    if _$compileDontDefineFunctions$Lisp then COMPILE(name)$Lisp
    name

makeFloatFunction(f, x, y) ==
    (u := mkLisp(convert(f)@INF)) case "failed" =>
      compiledFunction(f, x, y)
    name := mkDefun(u::INF, [ix := convert x, iy := convert y])
    t := [lsf, lsf]$List(INF)
    spadname := declare DI2
    spadform:=mkPretend convert([mkLispCall name,ix,iy]$List(INF))@INF
    interpret function(spadform, [x, y], spadname)
    binaryFunction compile(spadname, t)

makeFloatFunction(f, var) ==
    (u := mkLisp(convert(f)@INF)) case "failed" =>
      compiledFunction(f, var)
    name := mkDefun(u::INF, [ivar := convert var])
    t := [lsf]$List(INF)
    spadname := declare DI1
    spadform:= mkPretend convert([mkLispCall name,ivar]$List(INF))@INF
    interpret function(spadform, [var], spadname)
    unaryFunction compile(spadname, t)

mkLispList l ==
    ans := nil()$List(INF)
    for s in l repeat
      (u := mkLisp s) case "failed" => return "failed"
      ans := concat(u::INF, ans)
    reverse_! ans

mkLisp s ==
    atom? s => s
    op := first(l := destruct s)
    (u := mkLispList rest l) case "failed" => "failed"
    ll := u::List(INF)
streqlist?(op, "+","*/","-\]]) => convert(concat(op, ll))@INF
streq?(op, "+\") => gencode("EXPT", ll)
streqlist?(op, ["exp","sin","cos","tan","atan",
"log", "sinh","cosh","tanh","asinh","acosh","atanh","sqrt"])) =>
gencode(upperCase string symbol op, ll)
streq?(op, "nthRoot") =>
  second ll = convert(2::Integer)@INF =>gencode("SQRT",[first ll])
gencode("EXPT", concat(first ll, [1$INF / second ll]))
streq?(op, "float") =>
a := ll.1
e := ll.2
b := ll.3
_*(a, EXPT(b, e)$Lisp)$Lisp pretend INF
"failed"


package MKFUNC MakeFunction

)set break resume
)spool MakeFunction.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
expr := (x - exp x + 1)^2 * (sin(x^2) * x + 1)^3
--R
--R
--R (1)
--R 3 x 2 4 3 x 5 4 3 2 3
--R (x (%e ) + (- 2x - 2x )%e + x + 2x + x )sin(x )
--R +
--R 2 x 2 3 2 x 4 3 2 2 2
--R (3x (%e ) + (- 6x - 6x )%e + 3x + 6x + 3x )sin(x )
---R +
---R x 2  2  x  3  2  2  x  2
---R (3x \(\%e\) + (- 6x - 6x)\(\%e\) + 3x + 6x + 3x)\(\sin(x)\) + (\(\%e\))
---R +
---R x  2
---R (- 2x - 2)\(\%e\) + x + 2x + 1
---R

Type: Expression(Integer)
---E 1

---S 2 of 9

function(expr, f, x)
---R
---R
---R (2) f
---R
---R

Type: Symbol
---E 2

---S 3 of 9
tbl := [f(0.1 * i - 1) for i in 0..20]
---R
---R Compiling function f with type Float -> Float
---R
---R (3)
---R [0.0005391844 0362701574, 0.0039657551 1844206653,
---R 0.008545187 4833983689 2, 0.0116524883 210869965 06,
---R 0.0108618220 597822062 55, 0.005349542 8910500836 48,
---R 0.003249003 154979990 716, 0.0000233304 827609881 6001, 0.0,
---R 0.0000268186 878262599 4229, 0.00004691571 3720051642 621,
---R 0.0026924576 5968519586 08, 0.0101468681 7369135148 8,
---R 0.0313833725 8543810564 3, 0.0876991144 515465297 9,
---R 0.2313019789 3439968362, 0.5843743955 958098772, 1.4114930171 992819179,
---R 3.2216948276 75164252]
---R

Type: List(Float)
---E 3

---S 4 of 9
e := (x - y + 1)^2 * (x^2 * y + 1)^2
---R
---R
---R (4)
---R [4 4 5 4 2 3 6 5 4 3 2 2
---R x y + (- 2x - 2x + 2x )y + (x + 2x + x - 4x - 4x + 1)y
---R +
---R [4 3 2 2
---R (2x + 4x + 2x - 2x - 2)y + x + 2x + 1

Type: Polynomial(Integer)
---E 4

---S 5 of 9
function(e, g, [x, y])
--R
--R
--R (5) g
--R Type: Symbol
--E 5

-S 6 of 9
function(e, h, x, y)
--R
--R
--R (6) h
--R Type: Symbol
--E 6

-S 7 of 9
m1 := squareMatrix [[1, 2], [3, 4]]
--R
--R
--R +1 2+
--R (7) |
--R +3 4+
--R Type: SquareMatrix(2,Integer)
--E 7

-S 8 of 9
m2 := squareMatrix [[1, 0], [-1, 1]]
--R
--R
--R + 1 0+
--R (8) |
--R +- 1 1+
--R Type: SquareMatrix(2,Integer)
--E 8

-S 9 of 9
h(m1, m2)
--R
--R Compiling function h with type (SquareMatrix(2,Integer),SquareMatrix
--R (2,Integer)) -> SquareMatrix(2,Integer)
--R
--R ++ 7836 8960 +
--R (9) |
--R ++ 17132 19588+
--R Type: SquareMatrix(2,Integer)
--E 9
)spool
)lisp (bye)
CHAPTER 14. CHAPTER M

— MakeFunction.help —

====================================================================
MakeFunction examples
====================================================================

Tools for making interpreter functions from top-level expressions
Transforms top-level objects into interpreter functions.

It is sometimes useful to be able to define a function given by
the result of a calculation.

Suppose that you have obtained the following expression after several
computations and that you now want to tabulate the numerical values of
f for x between -1 and +1 with increment 0.1.

\[
\begin{align*}
\text{expr} & := (x - \exp x + 1)^2 \times (\sin(x^2) \times x + 1)^3 \\
& \quad \times (\exp(\exp x) + (-2x - 2x)\exp + x + 2x + x)\sin(x) \\
& \quad + 2 \times 2 \times 4 \times 3 \times 2 \times 4 \times 3 \times 2 \times 2 \\
& \quad \times (3\exp(x) + (-6x - 6x)\exp + 3x + 6x + 3x)\sin(x) \\
& \quad + x \times 2 \times 3 \times 3 \times 2 \times 3 \times 2 \times 2 \\
& \quad \times (3\exp(x) + (-6x - 6)\exp + 3x + 6x + 3x)\sin(x) + \exp \\
& \quad \times (-2x - 2)\exp + x + 2x + 1.
\end{align*}
\]

Type: Expression Integer

You could, of course, use the function eval within a loop and evaluate
expr twenty-one times, but this would be quite slow. A better way is
to create a numerical function f such that f(x) is defined by the
expression expr above, but without retyping expr! The package
MakeFunction provides the operation function which does exactly this.

Issue this to create the function f(x) given by expr.

\[
\text{function}(\text{expr}, f, x) \\
f
\]

Type: Symbol

To tabulate expr, we can now quickly evaluate f 21 times.

\[
\text{tbl} := [f(0.1 \times i - 1) \text{ for } i \text{ in } 0..20];
\]

Type: List Float

Use the list [x1,...,xn] as the third argument to function to create a
multivariate function f(x1,...,xn).
\[ e := (x - y + 1)^2 \cdot (x^2 \cdot y + 1)^2 \]
\[ x^4 + 4x^3 + 5x^2 + 4x + 3y + 2x + 3y + 2 \]
\[ + 4x + 3y + 2 \]
\[ + 2x + 4x + 2x - 2x - 2y + x + 2x + 1 \]
\[ \text{Type: Polynomial Integer} \]

\[ \text{function(e, g, [x, y])} \]
\[ g \]
\[ \text{Type: Symbol} \]

In the case of just two variables, they can be given as arguments without making them into a list.

\[ \text{function(e, h, x, y)} \]
\[ h \]
\[ \text{Type: Symbol} \]

Note that the functions created by function are not limited to floating point numbers, but can be applied to any type for which they are defined.

\[ m1 := \text{squareMatrix} [ [1, 2], [3, 4] ] \]
\[ +1 2+ \]
\[ +3 4+ \]
\[ \text{Type: SquareMatrix(2,Integer)} \]

\[ m2 := \text{squareMatrix} [ [1, 0], [-1, 1] ] \]
\[ +1 0+ \]
\[ +1 1+ \]
\[ \text{Type: SquareMatrix(2,Integer)} \]

\[ \text{h(m1, m2)} \]
\[ +7836 8960 + \]
\[ +17132 19588+ \]
\[ \text{Type: SquareMatrix(2,Integer)} \]

See Also:
\[ o )\text{show MakeFunction} \]
MakeFunction (MKFUNC)

Exports:
function

— package MKFUNC MakeFunction —

)abbrev package MKFUNC MakeFunction
++ Author: Manuel Bronstein
++ Date Created: 22 Nov 1988
++ Date Last Updated: 8 Jan 1990
++ Description:
++ Tools for making interpreter functions from top-level expressions
++ Transforms top-level objects into interpreter functions.

MakeFunction(S:ConvertibleTo InputForm): Exports == Implementation where
SY ==> Symbol

Exports ==> with
function: (S, SY ) -> SY
  ++ function(e, foo) creates a function \spad{foo() == e}.
function: (S, SY, SY) -> SY
  ++ function(e, foo, x) creates a function \spad{foo(x) == e}.
function: (S, SY, SY, SY) -> SY
  ++ function(e, foo, x, y) creates a function \spad{foo(x, y) = e}.
function: (S, SY, List SY) -> SY
  ++ \spad{function(e, foo, [x1,...,xn])} creates a function
  ++ \spad{foo(x1,...,xn) == e}.

Implementation ==> add
function(s, name) == function(s, name, nil())
function(s:S, name:SY, x:SY) == function(s, name, [x])
function(s, name, x, y) == function(s, name, [x, y])

function(s:S, name:SY, args:List SY) ==
  interpret function(convert s, args, name)$InputForm
  name
package MKRECORD MakeRecord

---

--- MakeRecord.input ---

)set break resume
)sys rm -f MakeRecord.output
)spool MakeRecord.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MakeRecord
--E 1

)spool
)lisp (bye)

---

--- MakeRecord.help ---

====================================================================
MakeRecord examples
====================================================================

MakeRecord is used internally by the interpreter to create record
types which are used for doing parallel iterations on streams.

See Also:
o )show MakeRecord
MakeRecord (MKRECORD)

Exports:
makeRecord

— package MKRECORD MakeRecord —

)abbrev package MKRECORD MakeRecord
++ Description:
++ MakeRecord is used internally by the interpreter to create record
++ types which are used for doing parallel iterations on streams.

MakeRecord(S: Type, T: Type): public == private where
public == with
    makeRecord: (S,T) -> Record(part1: S, part2: T)
    ++ makeRecord(a,b) creates a record object with type
    ++ Record(part1:S, part2:T),
    ++ where part1 is \spad{a} and part2 is \spad{b}.
private == add
    makeRecord(s: S, t: T) ==
        [s,t]$Record(part1: S, part2: T)

— MKRECORD.dotabb —

"MKRECORD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKRECORD"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKUCFUNC" -> "KONVERT"
"MKUCFUNC" -> "TYPE"
package MKUCFUNC MakeUnaryCompiledFunction

---

MakeUnaryCompiledFunction.input ---

(setq break resume
(sys rm -f MakeUnaryCompiledFunction.output
(spool MakeUnaryCompiledFunction.output
(set message test on
(set message auto off
(clear all

--S 1 of 1
(show MakeUnaryCompiledFunction
--E 1

(spool
(lisp (bye)

---

MakeUnaryCompiledFunction.help ---

====================================================================
MakeUnaryCompiledFunction examples
====================================================================

Tools for making compiled functions from top-level expressions
Transforms top-level objects into compiled functions.

See Also:
 o )show MakeUnaryCompiledFunction

---
MakeUnaryCompiledFunction (MKUCFUNC)

Exports:
- unaryFunction
- compiledFunction

--- package MKUCFUNC MakeUnaryCompiledFunction ---

)abbrev package MKUCFUNC MakeUnaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools for making compiled functions from top-level expressions
++ Transforms top-level objects into compiled functions.

MakeUnaryCompiledFunction(S, D, I): Exports == Implementation where
  S: ConvertibleTo InputForm
  D, I: Type

SY ==> Symbol
DI ==> devaluate(D -> I)$Lisp

Exports ==> with
  unaryFunction : SY -> (D -> I)
  ++ unaryFunction(a) is a local function
  compiledFunction: (S, SY) -> (D -> I)
  ++ compiledFunction(expr, x) returns a function \spad{f: D -> I}
  ++ defined by \spad{f(x) == expr}.
  ++ Function f is compiled and directly
  ++ applicable to objects of type D.

Implementation ==> add
  import MakeFunction(S)

  func: (SY, D) -> I

  func(name, x) == FUNCALL(name, x, NIL$Lisp)$Lisp
unaryFunction name == (d1:D):I +-> func(name, d1)

compiledFunction(e:S, x:SY) ==
  t := [convert([devaluate(D)$Lisp]$List(InputForm))
   ]$List(InputForm)
  unaryFunction compile(function(e, declare DI, x), t)

——

— MKUCFUNC.dotabb —
"MKUCFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKUCFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKUCFUNC" -> "KONVERT"
"MKUCFUNC" -> "TYPE"

——

package MAPHACK1 MappingPackageInternalHacks1

— MappingPackageInternalHacks1.input —

)set break resume
)sys rm -f MappingPackageInternalHacks1.output
)spool MappingPackageInternalHacks1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MappingPackageInternalHacks1
--E 1

)spool
)lisp (bye)

——

— MappingPackageInternalHacks1.help —

====================================================================
MappingPackageInternalHacks1 examples
Various Currying operations.

See Also:
o )show MappingPackageInternalHacks1

---

MappingPackageInternalHacks1 (MAPHACK1)

Exports:
iter recur

--- package MAPHACK1 MappingPackageInternalHacks1 ---

)abbrev package MAPHACK1 MappingPackageInternalHacks1
++ Author: S.M.Watt and W.H.Burge
++ Date Created:Jan 87
++ Date Last Updated:Feb 92
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks1(A: SetCategory): MPcat == MPdef where
  NNI ==> NonNegativeInteger

MPcat == with
  iter: ((A -> A), NNI, A) -> A
  ++\spad{iter(f,n,x)} applies \spad{f n} times to \spad{x}.
  recur: ((NNI, A)->A, NNI, A) -> A
  ++\spad{recur(n,g,x)} is \spad{g(n,g(n-1,..g(1,x)..))}.)

MPdef == add
iter(g,n,x) ==
  for i in 1..n repeat x := g x -- g(g(..(x)..))
x
recur(g,n,x) ==
  for i in 1..n repeat x := g(i,x) -- g(n,g(n-1,..g(1,x)..))
x

package MAPHACK2 MappingPackageInternalHacks2

-- MappingPackageInternalHacks2.input --

)set break resume
)sys rm -f MappingPackageInternalHacks2.output
)spool MappingPackageInternalHacks2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MappingPackageInternalHacks2
--E 1

)spool
)lisp (bye)

-- MappingPackageInternalHacks2.help --

====================================================================
MappingPackageInternalHacks2 examples
Various Currying operations.

See Also:
  o )show MappingPackageInternalHacks2

---

**MappingPackageInternalHacks2 (MAPHACK2)**

Exports:
  arg1  arg2

--- package MAPHACK2 MappingPackageInternalHacks2 ---

)abbrev package MAPHACK2 MappingPackageInternalHacks2
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks2(A: SetCategory, C: SetCategory):_
  MPcat == MPdef where
    NNI ==> NonNegativeInteger

  MPcat == with
    arg1: (A, C) -> A
    ++\spad{arg1(a,c)} selects its first argument.
    arg2: (A, C) -> C
    ++\spad{arg2(a,c)} selects its second argument.

  MPdef == add
    arg1(a, c)  == a
    arg2(a, c)  == c
package MAPHACK3 MappingPackageInternalHacks3

---

--- MAPHACK2.dotabb ---

"MAPHACK2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPHACK2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPHACK2" -> "BASTYPE"
"MAPHACK2" -> "KOERCE"

---

package MAPHACK3 MappingPackageInternalHacks3

--- MappingPackageInternalHacks3.input ---

)set break resume
)sys rm -f MappingPackageInternalHacks3.output
)spool MappingPackageInternalHacks3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MappingPackageInternalHacks3
--E 1

)spool
)lisp (bye)

---

--- MappingPackageInternalHacks3.help ---

====================================================================
MappingPackageInternalHacks3 examples
====================================================================

Various Currying operations.

See Also:
  o )show MappingPackageInternalHacks3

---
MappingPackageInternalHacks3 (MAPHACK3)

Exports:

comp

— package MAPHACK3 MappingPackageInternalHacks3 —

)abbrev package MAPHACK3 MappingPackageInternalHacks3
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks3(A: SetCategory, B: SetCategory, C: SetCategory):_
MPcat == MPdef where
  NNI ==> NonNegativeInteger

MPcat == with
  comp: (B->C, A->B, A) -> C
  \spad{comp(f,g,x)} is \spad{f(g(x))}.

MPdef == add
  comp(g,h,x) == g h x

— MAPHACK3.dotabb —
package MAPPKG1 MappingPackage1

--- MAPPINGPACKAGE1.input ---

)set break resume
)spool MappingPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R (Integer) has been added to workspace.
--R
--E 1

--S 2 of 26
power(2,3)
--R
--R Compiling function power with type (Fraction(Integer),Integer) ->
--R Fraction(Integer)
--R
--R (2) 8
--R
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R (3) theMap(MAPPKG3;twist;MM;5!0)
--R
--R Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
--R
--R
--R (4) 8
--R
--E 4
square: FRAC INT → FRAC INT

square := curryRight(power, 2)

squirrel := constantRight(square) $ MAPPKG3 (FRAC INT, FRAC INT, FRAC INT)

sixteen := curry(square, 4/1)
sixteen()
   Type: Fraction(Integer)

square2 := square * square
   Type: Fraction(Integer)

square2 3
   Type: Fraction(Integer)

sc(x: FRAC INT): FRAC INT == x + 1
   Type: Void

incfns := [sc**i for i in 0..10]
   Type: List((Fraction(Integer) -> Fraction(Integer)))

[f 4 for f in incfns]
times(n:NNI, i:INT):INT == n*i

Function declaration times: (NonNegativeInteger,Integer) -> Integer has been added to workspace.

r := recur(times)

Compiling function times with type (NonNegativeInteger,Integer) -> Integer

theMap(MAPPKG1;recur;2M;7!0,0)

Type: ((NonNegativeInteger,Integer) -> Integer)

fact := curryRight(r, 1)

theMap(MAPPKG3;curryRight;MBM;1!0,0)

Type: (NonNegativeInteger -> Integer)

mto2ton(m, n) == 
  raiser := square^n
  raiser m

mto2ton(3, 3)
--R Compiling function mto2ton with type (PositiveInteger, PositiveInteger) -> Fraction(Integer)
--R
--R (22) 6561
--R Type: Fraction(Integer)
--E 22

--S 23 of 26
shiftfib(r: List INT) : INT ==
t := r.1
r.1 := r.2
r.2 := r.2 + t
t
--R Function declaration shiftfib : List(Integer) -> Integer has been added to workspace.
--R Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R (24) [0,1]
--R Type: List(Integer)
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
--R Compiling function shiftfib with type List(Integer) -> Integer
--R
--R (25) theMap(MAPPKG2;curry;MAM;2!0,0)
--R Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R (26)
--R [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R 317811, 514229, 832040]
--R Type: List(Integer)
--E 26

)spool
)lisp (bye)
Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate some library operations from the packages MappingPackage1, MappingPackage2, and MappingPackage3 that manipulate and create functions. Some terminology: a nullary function takes no arguments, a unary function takes one argument, and a binary function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

\[
power(q: \text{FRAC INT}, n: \text{INT}): \text{FRAC INT} = q^n\]

\[
power(2,3)\]

\[
\text{Type: Fraction Integer}\]

The twist operation transposes the arguments of a binary function. Here \(\text{rewop}(a, b) = \text{power}(b, a)\).

\[
\text{rewop} := \text{twist power}\]

\[
\text{theMap(MAPPKG3;twist;MM;5!0)}\]

\[
\text{Type: ((Integer,Fraction Integer) \to Fraction Integer)}\]

This is \(2^3\).

\[
\text{rewop}(3, 2)\]

\[
8\]

\[
\text{Type: Fraction Integer}\]

Now we define square in terms of power.

\[
square: \text{FRAC INT} \to \text{FRAC INT}\]

\[
\text{Type: Void}\]

The curryRight operation creates a unary function from a binary one by providing a constant argument on the right.

\[
square := \text{curryRight(power, 2)}\]

\[
\text{theMap(MAPPKG3;curryRight;MM;1!0,0)}\]

\[
\text{Type: (Fraction Integer \to Fraction Integer)}\]
Likewise, the curryLeft operation provides a constant argument on the left.

```
    square 4
    16
    Type: Fraction Integer
```

The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that g(a,b) = f(a).

```
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
theMap(MAPPKG3;constantRight;MM;3!0)
    Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)
```

Likewise, constantLeft(f) is the function g such that g(a,b) = f(b).

```
squirrel(1/2, 1/3)
    1
    -
    4
    Type: Fraction Integer
```

The curry operation makes a unary function nullary.

```
sixteen := curry(square, 4/1)
    theMap(MAPPKG2;curry;MAM;2!0,0)
    Type: (() -> Fraction Integer)
```

```
sixteen()
    16
    Type: Fraction Integer
```

The * operation constructs composed functions.

```
square2:=square*square
    theMap(MAPPKG3;*;MMM;6!0,0)
    Type: (Fraction Integer -> Fraction Integer)
```

```
square2 3
    81
    Type: Fraction Integer
```

Use the ** operation to create functions that are n-fold iterations of other functions.

```
sc(x: FRAC INT): FRAC INT == x + 1
    Type: Void
```
This is a list of Mapping objects.

\[
\text{incfns := } [\text{sc**i for i in 0..10}]
\text{[theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),}
\text{theMap(MAPPKG1;**;MNniM;6!0,0)]}
\text{Type: List (Fraction Integer -> Fraction Integer)}
\]

This is a list of applications of those functions.

\[
[f 4 for f in incfns]
[4,5,6,7,8,9,10,11,12,13,14]
\text{Type: List Fraction Integer}
\]

Use the recur operation for recursion:

\[
g := \text{recur } f \text{ means } g(n,x) == f(n,f(n-1,...f(1,x))).
\]

\[
times(n:NNI, i:INT):INT == n*i
\text{Type: Void}
\]

\[
r := \text{recur(times)}
\text{theMap(MAPPKG1;recur;2M;7!0,0)}
\text{Type: ((NonNegativeInteger,Integer) -> Integer)}
\]

This is a factorial function.

\[
\text{fact := curryRight(r, 1)}
\text{theMap(MAPPKG3;curryRight;MBM;1!0,0)}
\text{Type: (NonNegativeInteger -> Integer)}
\]

\[
\text{fact 4}
24
\text{Type: PositiveInteger}
\]

Constructed functions can be used within other functions.

\[
mto2ton(m, n) ==
\text{raiser := square"n}
\text{raiser m}
\text{Type: Void}
\]

This is 3^{(2^3)}.

\[
mto2ton(3, 3)
6561
\text{Type: Fraction Integer}
\]
Here shiftfib is a unary function that modifies its argument.

```lisp
shiftfib(r: List INT) : INT ==
t := r.1
r.1 := r.2
r.2 := r.2 + t
t
Type: Void
```

By currying over the argument we get a function with private state.

```lisp
fibinit: List INT := [0, 1]
[0,1]
Type: List Integer

fibs := curry(shiftfib, fibinit)
theMap(MAPPKG2;curry;MAM;2!0,0)
Type: (() -> Integer)

[fibs() for i in 0..30]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
317811, 514229, 832040]
Type: List Integer
```

See Also:
- `)show MappingPackage1`
- `)help MappingPackage2`
- `)help MappingPackage3`
- `)help MappingPackage4`
MappingPackage1 (MAPPKG1)

Exports:
coerce  fixedPoint  id  nullary  recur  ?**?

— package MAPPKG1 MappingPackage1 —

)abbrev package MAPPKG1 MappingPackage1
++ Author: S.M.Watt and W.H.Burge
++ Date Created:Jan 87
++ Date Last Updated:Feb 92
++ Description:
++ Various Currying operations.

MappingPackage1(A:SetCategory): MPcat == MPdef where
  NNI  ==>  NonNegativeInteger

MPcat == with
  nullary: A   ->  (A->A)
    +++ \spad{nullary A} changes its argument into a
    +++ nullary function.
  coerce: A   ->  (A->A)
    +++ \spad{coerce A} changes its argument into a
    +++ nullary function.
  fixedPoint: (A->A)   ->  A
    +++ \spad{fixedPoint f} is the fixed point of function \spad{f}.
    +++ that is, such that \spad{fixedPoint f = f(fixedPoint f)}.
  fixedPoint: (List A->List A, Integer)   ->  List A
    +++ \spad{fixedPoint(f,n)} is the fixed point of function
    +++ \spad{f} which is assumed to transform a list of length
    +++ \spad{n}.
  id: A   ->  A
    +++ \spad{id x} is \spad{x}.
"**": (A->A, NNI) -> (A->A)
++\texttt{spad{f**n}} is the function which is the n-fold application
++ of \texttt{spad{f}}.
recur: ((NNI, A)->A) -> ((NNI, A)->A)
++\texttt{spad{recur(g)}} is the function \texttt{spad{h}} such that
++ \texttt{spad{h(n,x)= g(n,g(n-1,...g(1,x)..))))}.

MPdef == add

MappingPackageInternalHacks1(A)
a: A
faa: A -> A
f0a: ()-> A
nullary a == a
coerce a == nullary a
fixedPoint faa ==
g0 := GENSYM()$Lisp
g1 := faa g0
EQ(g0, g1)$Lisp => error "All points are fixed points"
GEQNSUBSTLIST([g0]$Lisp, [g1]$Lisp, g1)$Lisp

fixedPoint(fll, n) ==
g0 := [(GENSYM()$Lisp):A for i in 1..n]
g1 := fll g0
or/[EQ(e0,e1)$Lisp for e0 in g0 for e1 in g1] =>
error "All points are fixed points"
GEQNSUBSTLIST(g0, g1, g1)$Lisp

-- Composition and recursion.
id a == a
g**n == (a1:A):A +-> iter(g, n, a1)
recur fnaa == (n1:NNI,a2:A):A +-> recur(fnaa, n1, a2)
package MAPPKG2 MappingPackage2

— MappingPackage2.input —

)set break resume
)spool MappingPackage2.output
)set message test on
)set message auto off
)clear all
--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R (Integer) has been added to workspace.
--R Type: Void
--E 1

--S 2 of 26
power(2,3)
--R
--R Compiling function power with type (Fraction(Integer),Integer) -> Fraction
--R Fraction(Integer)
--R (2) 8
--R Type: Fraction(Integer)
--E 2

--S 3 of 26
rewop := twist power
--R
--R (3) theMap(MAPPKG3;twist;MM;5!0)
--R Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
--R
--R (4) 8
--R Type: Fraction(Integer)
--E 4
--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R
--R Type: Void
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R (6) theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R Type: (Fraction(Integer) -> Fraction(Integer))
--E 6

--S 7 of 26
square 4
--R
--R
--R (7) 16
--R Type: Fraction(Integer)
--E 7

--S 8 of 26
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R (8) theMap(MAPPKG3;constantRight;MM;3!0)
--R Type: ((Fraction(Integer),Fraction(Integer)) -> Fraction(Integer))
--E 8

--S 9 of 26
squirrel(1/2, 1/3)
--R
--R
--R 1
--R (9) -
--R 4
--R Type: Fraction(Integer)
--E 9

--S 10 of 26
sixteen := curry(square, 4/1)
--R
--R
--R (10) theMap(MAPPKG2;curry;MAM;2!0,0)
--R Type: (() -> Fraction(Integer))
--E 10

--S 11 of 26
sixteen()
1488

--R
--R
--R (11) 16
--R
--E 11

--S 12 of 26
square2:=square*square
--R
--R
--R (12) theMap(MAPPKG3;*;MMM;6!0,0)
--R Type: (Fraction(Integer) -> Fraction(Integer))
--E 12

--S 13 of 26
square2 3
--R
--R
--R (13) 81
--R Type: Fraction(Integer)
--E 13

--S 14 of 26
sc(x: FRAC INT): FRAC INT == x + 1
--R
--R Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
--R been added to workspace.
--R Type: Void
--E 14

--S 15 of 26
incfns := [sc**i for i in 0..10]
--R
--R Compiling function sc with type Fraction(Integer) -> Fraction(
--R Integer)
--R
--R (15)
--R [theMap(MAPPKG1;**;MMniM;6!0,0), theMap(MAPPKG1;**;MMniM;6!0,0),
--R theMap(MAPPKG1;**;MMniM;6!0,0), theMap(MAPPKG1;**;MMniM;6!0,0),
--R theMap(MAPPKG1;**;MMniM;6!0,0), theMap(MAPPKG1;**;MMniM;6!0,0),
--R theMap(MAPPKG1;**;MMniM;6!0,0), theMap(MAPPKG1;**;MMniM;6!0,0),
--R theMap(MAPPKG1;**;MMniM;6!0,0), theMap(MAPPKG1;**;MMniM;6!0,0)]
--R Type: List((Fraction(Integer) -> Fraction(Integer)))
--E 15

--S 16 of 26
[f 4 for f in incfns]
--R
--R
times(n:NNI, i:INT):INT == n*i
--R Function declaration times : (NonNegativeInteger, Integer) -> Integer
--R has been added to workspace.
--E 17

r := recur(times)
--R Compiling function times with type (NonNegativeInteger, Integer) -> Integer
--E 18

fact := curryRight(r, 1)
--R
--E 19

mto2ton(m, n) ==
  raiser := square^n m
--R
--E 20

mto2ton(3, 3)
--R PositiveInteger) -> Fraction(Integer)
--R
--R (22) 6561
--R
--E 22

--S 23 of 26
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R
Function declaration shiftfib : List(Integer) -> Integer has been
--R added to workspace.
Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R
--R (24) [0,1]
--R
--R
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
--R Compiling function shiftfib with type List(Integer) -> Integer
--R
--R (25) theMap(MAPPKG2;curry;MAM;2!0,0)
--R
Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R
--R (26)
--R [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R 317811, 514229, 832040]
--R
Type: List(Integer)
--E 26

)spool
)lisp (bye)
MappingPackage2.help

MappingPackage examples

Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate some library operations from the packages MappingPackage1, MappingPackage2, and MappingPackage3 that manipulate and create functions. Some terminology: a nullary function takes no arguments, a unary function takes one argument, and a binary function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

\[
power(q: \text{FRAC INT}, n: \text{INT}): \text{FRAC INT} = q^{*n}
\]

\[
power(2, 3)
\]

\[
8
\]

The twist operation transposes the arguments of a binary function. Here rewop(a, b) is power(b, a).

\[
rewop := \text{twist power}
\]

\[
\text{theMap(MAPPGK3; twist; MM; 5!0)}
\]

\[
\text{Type: ((Integer, Fraction Integer) -> Fraction Integer)}
\]

This is 2^3.

\[
rewop(3, 2)
\]

\[
8
\]

Now we define square in terms of power.

\[
square: \text{FRAC INT} -> \text{FRAC INT}
\]

\[
\text{Type: Void}
\]

The curryRight operation creates a unary function from a binary one by providing a constant argument on the right.

\[
square := \text{curryRight(power, 2)}
\]

\[
\text{theMap(MAPPKG3; curryRight; MBM; 1!0, 0)}
\]

\[
\text{Type: (Fraction Integer -> Fraction Integer)}
\]

Likewise, the curryLeft operation provides a constant argument on the
The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that g(a,b) = f(a).

\[\text{squirrel} := \text{constantRight(square)} \]\n
\[\text{theMap(MAPPKG3;constantRight;MM;3!0)}\]

Type: \((\text{Fraction Integer},\text{Fraction Integer}) \rightarrow \text{Fraction Integer}\)

Likewise, constantLeft(f) is the function g such that g(a,b) = f(b).

\[\text{squirrel}(1/2, 1/3)\]

\[1 - \frac{1}{4}\]

Type: Fraction Integer

The curry operation makes a unary function nullary.

\[\text{sixteen} := \text{curry(square, 4/1)}\]

\[\text{theMap(MAPPKG2;curry;MAM;2!0,0)}\]

Type: \((\text{} \rightarrow \text{Fraction Integer})\)

\[\text{sixteen}()\]

\[16\]

Type: Fraction Integer

The * operation constructs composed functions.

\[\text{square2} := \text{square*square}\]

\[\text{theMap(MAPPKG3;*;MMM;6!0,0)}\]

Type: \((\text{Fraction Integer} \rightarrow \text{Fraction Integer})\)

\[\text{square2} 3\]

\[81\]

Type: Fraction Integer

Use the ** operation to create functions that are \(n\)-fold iterations of other functions.

\[\text{sc}(x: \text{FRAC INT}): \text{FRAC INT} \equiv x + 1\]

Type: Void

This is a list of Mapping objects.
incfns := [sc**i for i in 0..10]
[theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
 theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
 theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
 theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
 theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
 theMap(MAPPKG1;**;MNniM;6!0,0)]
Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

[f 4 for f in incfns]
[4,5,6,7,8,9,10,11,12,13,14]
Type: List Fraction Integer

Use the recur operation for recursion:

g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).
times(n:NNI, i:INT):INT == n*i
Type: Void
r := recur(times)
theMap(MAPPKG1;recur;2M;7!0,0)
Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

fact := curryRight(r, 1)
theMap(MAPPKG3;curryRight;MBM;1!0,0)
Type: (NonNegativeInteger -> Integer)

fact 4
24
Type: PositiveInteger

Constructed functions can be used within other functions.

mto2ton(m, n) ==
raiser := square^n
raiser m
Type: Void

This is 3**(2^3).
mto2ton(3, 3)
6561
Type: Fraction Integer

Here shiftfib is a unary function that modifies its argument.
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t

Type: Void

By currying over the argument we get a function with private state.

fibinit: List INT := [0, 1]

Type: List Integer

fibs := curry(shiftfib, fibinit)

theMap(MAPPKG2; curry; MAM; 2!0, 0)

Type: () -> Integer

[fibs() for i in 0..30]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040]

Type: List Integer

See Also:
o )help MappingPackage1
o )show MappingPackage2
o )help MappingPackage3
o )help MappingPackage4

MappingPackage2 (MAPPKG2)
Exports:
const constant curry diag

— package MAPPKG2 MappingPackage2 —

)abbrev package MAPPKG2 MappingPackage2
++ Description:
++ Various Currying operations.

MappingPackage2(A:SetCategory, C:SetCategory): MPcat == MPdef where
  NNI   => NonNegativeInteger

MPcat == with
  const: C -> (A ->C)
  ++ \spad{const c} is a function which produces \spad{c} when++ applied to its argument.
  curry: (A ->C, A) -> (()->C)
  ++ \spad{cu(f,a)} is the function \spad{g}++ such that \spad{g ()= f a}..
  constant: (()->C) -> (A ->C)
  ++ \spad{vu(f)} is the function \spad{g}++ such that \spad{g a= f ()}.
  diag: ((A,A)->C) -> (A->C)
  ++ \spad{diag(f)} is the function \spad{g}++ such that \spad{g a = f(a,a)}..

MPdef == add

MappingPackageInternalHacks2(A, C)

a: A
c: C
faa: A -> A
f0c: ()-> C
fac: A -> C
faac: (A,A)->C

const c == (a1:A):C +-> arg2(a1, c)
curry(fac, a) == fac a
constant f0c == (a1:A):C +-> arg2(a1, f0c())
diag faac == (a1:A):C +-> faac(a1, a1)

— MAPPKG2.dotabb —
package MAPPKG3 MappingPackage3

--- MappingPackage3.input ---

)set break resume
)spool MappingPackage3.output
)set message test on
)set message auto off
)clear all
--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R (Integer) has been added to workspace.
--R
--E 1

--S 2 of 26
power(2,3)
--R
--R Compiling function power with type (Fraction(Integer),Integer) -> Fraction
--R  Fraction(Integer)
--R
--R (2) 8
--R
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R (3) theMap(MAPPKG3;twist;MM;5!0)
--R  Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
--R
--R
--R (4) 8
--R
--E 4

--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R (6) theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R
--E 6

--S 7 of 26
square 4
--R
--R
--R (7) 16
--R
--E 7

--S 8 of 26
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R (8) theMap(MAPPKG3;constantRight;MM;3!0)
--R
--E 8

--S 9 of 26
squirrel(1/2, 1/3)
--R
--R
--R 1
--R (9) -
--R 4
--R
--E 9

--S 10 of 26
sixteen := curry(square, 4/1)
--R
--R
--R (10) theMap(MAPPKG2;curry;MAM;2!0,0)


--R Type: ((() -> Fraction(Integer))
--E 10

--S 11 of 26
sixteen()
--R
--R
--R (11) 16
--R
--E 11

--S 12 of 26
square2 := square * square
--R
--R
--R (12) theMap(MAPPKG3;*;MMM;6!0,0)
--R
--E 12

--S 13 of 26
square2 3
--R
--R
--R (13) 81
--R
--E 13

--S 14 of 26
sc(x: FRAC INT): FRAC INT == x + 1
--R
--R Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
--R been added to workspace.
--R
--E 14

--S 15 of 26
incfns := [sc**i for i in 0..10]
--R
--R Compiling function sc with type Fraction(Integer) -> Fraction( Integer)
--R
--R (15)
--R [theMap(MAPPKG1;++;MNNMi;6!0,0), theMap(MAPPKG1;++;MNNMi;6!0,0),
--R theMap(MAPPKG1;++;MNNMi;6!0,0), theMap(MAPPKG1;++;MNNMi;6!0,0),
--R theMap(MAPPKG1;++;MNNMi;6!0,0), theMap(MAPPKG1;++;MNNMi;6!0,0),
--R theMap(MAPPKG1;++;MNNMi;6!0,0), theMap(MAPPKG1;++;MNNMi;6!0,0),
--R theMap(MAPPKG1;++;MNNMi;6!0,0), theMap(MAPPKG1;++;MNNMi;6!0,0),
--R theMap(MAPPKG1;++;MNNMi;6!0,0)]
--E 15
[f 4 for f in incfns]

\[ \text{Type: List(Fraction(Integer))} \]

\[ \text{Type: Void} \]

\[ \text{Type: ((NonNegativeInteger,Integer) -> Integer)} \]

\[ \text{Type: PositiveInteger} \]

\[ \text{Type: Void} \]
mto2ton(3, 3)

Compiling function mto2ton with type (PositiveInteger, PositiveInteger) -> Fraction(Integer)

mto2ton(3, 3) 6561

Type: Fraction(Integer)

shiftfib(r: List INT) : INT ==
    t := r.1
    r.1 := r.2
    r.2 := r.2 + t
    t

Function declaration shiftfib : List(Integer) -> Integer has been added to workspace.

Type: Void

fibinit: List INT := [0, 1]

[0,1]

Type: List(Integer)

fibs := curry(shiftfib, fibinit)

Compiling function shiftfib with type List(Integer) -> Integer

theMap(MAPPKG2;curry;MAM;2!0,0)

Type: (() -> Integer)

[fibs() for i in 0..30]

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040]
package mappingpackage3

---

MappingPackage3.help

====================================================================
MappingPackage examples
====================================================================

Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate
some library operations from the packages MappingPackage1, MappingPackage2,
and MappingPackage3 that manipulate and create functions. Some terminology:
- A nullary function takes no arguments, a unary function takes one argument,
- and a binary function takes two arguments.

We begin by creating an example function that raises a rational number
to an integer exponent.

\[
\text{power}(q: \text{FRAC INT}, n: \text{INT}) : \text{FRAC INT} = q^n
\]

\[
\text{power}(2, 3)
\]

\[
8
\]

The twist operation transposes the arguments of a binary function.
Here \( \text{rewop}(a,b) \) is \( \text{power}(b,a) \).

\[
\text{rewop} := \text{twist} \text{power}
\]

\[
\text{theMap}(\text{MAPPKG3}; \text{twist}; \text{MM}; 5\!0)
\]

\[
\text{Type: ((Integer, Fraction Integer) \to Fraction Integer)}
\]

This is \( 2^3 \).

\[
\text{rewop}(3, 2)
\]

\[
8
\]

Now we define square in terms of power.

\[
\text{square} : \text{FRAC INT} \to \text{FRAC INT}
\]

\[
\text{Type: Void}
\]

The curryRight operation creates a unary function from a binary one by
providing a constant argument on the right.
square := curryRight(power, 2)  
theMap(MAPPKG3; curryRight; MBM; 1!0, 0)  
Type: (Fraction Integer -> Fraction Integer)

Likewise, the curryLeft operation provides a constant argument on the left.

square 4  
16  
Type: Fraction Integer

The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that g(a,b) = f(a).

squirrel := constantRight(square)$MAPPKG3(FRAC INT, FRAC INT, FRAC INT)  
theMap(MAPPKG3; constantRight; MM; 3!0)  
Type: ((Fraction Integer, Fraction Integer) -> Fraction Integer)

Likewise, constantLeft(f) is the function g such that g(a,b) = f(b).

squirrel(1/2, 1/3)  
1  
-  
4  
Type: Fraction Integer

The curry operation makes a unary function nullary.

sixteen := curry(square, 4/1)  
theMap(MAPPKG2; curry; MAM; 2!0, 0)  
Type: (() -> Fraction Integer)

sixteen()  
16  
Type: Fraction Integer

The * operation constructs composed functions.

square2 := square * square  
theMap(MAPPKG3; *; MMM; 6!0, 0)  
Type: (Fraction Integer -> Fraction Integer)

square2 3  
81  
Type: Fraction Integer

Use the ** operation to create functions that are n-fold iterations of other functions.
sc(x: FRAC INT): FRAC INT == x + 1
Type: Void

This is a list of Mapping objects.

incfns := [sc**i for i in 0..10]
[theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
theMap(MAPPKG1;**;MNniM;6!0,0)]
Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

[f 4 for f in incfns]
[4,5,6,7,8,9,10,11,12,13,14]
Type: List Fraction Integer

Use the recur operation for recursion:

g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).
times(n:NNI, i:INT):INT == n*i
Type: Void

r := recur(times)
theMap(MAPPKG1;recur;2M;7!0,0)
Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

fact := curryRight(r, 1)
theMap(MAPPKG3;curryRight;MBM;110,0)
Type: (NonNegativeInteger -> Integer)

fact 4
24
Type: PositiveInteger

Constructed functions can be used within other functions.

mto2ton(m, n) ==
raiser := square^n
raiser m
Type: Void

This is 3^{2^3}.
Here shiftfib is a unary function that modifies its argument.

```
shiftfib(r: List INT) : INT ==
t := r.1
r.1 := r.2
r.2 := r.2 + t
t
```

By currying over the argument we get a function with private state.

```
fibinit: List INT := [0, 1]
fibs := curry(shiftfib, fibinit)
theMap(MAPPKG2;curry;MAM;2!0,0)
[fibs() for i in 0..30]
```

See Also:
- `)help MappingPackage1`
- `)help MappingPackage2`
- `)show MappingPackage3`
- `)help MappingPackage4`
MappingPackage3 (MAPPKG3)

Exports:
constantLeft constantRight curryLeft curryRight twist **?

— package MAPPKG3 MappingPackage3 —

)abbrev package MAPPKG3 MappingPackage3
++ Description:
++ Various Currying operations.

MappingPackage3(A:SetCategory, B:SetCategory, C:SetCategory):_
MPcat == MPdef where
  NNI === NonNegativeInteger

MPcat == with
  curryRight: ((A,B)->C, B) -> (A ->C)
  
  curryLeft: ((A,B)->C, A) -> (B ->C)
  
  constantRight: (A -> C) -> ((A,B)->C)
  
  constantLeft: (B -> C) -> ((A,B)->C)
  
  twist: ((A,B)->C) -> ((B,A)->C)
  
  "**": (B->C, A->B) -> (A->C)

++ \spad{curryRight(f,b)} is the function \spad{g} such that ++ \spad{g a = f(a,b)}.
++ \spad{curryLeft(f,a)} is the function \spad{g} ++ such that \spad{g b = f(a,b)}.
++ \spad{constantRight(f)} is the function \spad{g} ++ such that \spad{g (a,b)= f a}.
++ \spad{constantLeft(f)} is the function \spad{g} ++ such that \spad{g (a,b)= f b}.
++ \spad{twist(f)} is the function \spad{g} ++ such that \spad{g (a,b)= f(b,a)}.
++ \spad{f*g} is the function \spad{h} ++ such that \spad{h x= f(g x)}. 
\[MP\text{def} == \text{add}\]

\begin{verbatim}
MappingPackageInternalHacks3(A, B, C)

a: A
b: B
c: C
faa: A -> A
f0c: () -> C
fac: A -> C
fbc: B -> C
fab: A -> B
fabc: (A,B) -> C
faac: (A,A) -> C

-- Fix left and right arguments as constants.
curryRight(fabc, b) == (a:A): C +-> fabc(a,b)
curryLeft(fabc, a) == (b:B): C +-> fabc(a,b)

-- Add left and right arguments which are ignored.
constantRight fac == (a:A, b:B): C +-> fac a
constantLeft fbc == (a:A, b:B): C +-> fbc b

-- Combinators to rearrange arguments.
twist fabc == (b:B, a:A): C +-> fabc(a,b)

-- Functional composition
fbc*fab == (a:A): C +-> \text{comp}(fbc, fab, a)
\end{verbatim}

\[\text{MAPPKG3.dotabb}\]
package MAPPKG4 MappingPackage4

— MappingPackage4.input —

)set break resume
)set message test on
)set message auto off
)clear all

--S 1 of 21
p:=(x:EXPR(INT)):EXPR(INT)+->3*x
--R
--R (1) theMap(Closure)
--R Type: (Expression Integer -> Expression Integer)
--E 1

--S 2 of 21
q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
--R
--R (2) theMap(Closure)
--R Type: (Expression Integer -> Expression Integer)
--E 2

--S 3 of 21
(p+q)(4)-(p(4)+q(4))
--R
--R (3) 0
--R Type: Expression Integer
--E 3

--S 4 of 21
(p+q)(x)-(p(x)+q(x))
--R
--R (4) 0
--R Type: Expression Integer
--E 4

--S 5 of 21
(p-q)(4)-(p(4)-q(4))
--R
--R (5) 0
--R Type: Expression Integer
\[(p - q)(x) - (p(x) - q(x))\]

Type: Expression Integer

\[(p \times q)(4) - (p(4) \times q(4))\]

Type: Expression Integer

\[(p \times q)(x) - (p(x) \times q(x))\]

Type: Expression Integer

\[(p/q)(4) - (p(4)/q(4))\]

Type: Expression Integer

\[(p/q)(x) - (p(x)/q(x))\]

Type: Expression Integer

\[r := (x: \text{INT}) : \text{INT} \rightarrow (x \times x)\]

Type: (Integer -> Integer)
--S 12 of 21
s:=(y:INT):INT++> (y*y+3)
--R
--R
--R (12) theMap(Closure)
--R Type: (Integer -> Integer)
--E 12

--S 13 of 21
(r+s)(4)-(r(4)+s(4))
--R
--R
--R (13) 0
--R Type: NonNegativeInteger
--E 13

--S 14 of 21
(r-s)(4)-(r(4)-s(4))
--R
--R
--R (14) 0
--R Type: NonNegativeInteger
--E 14

--S 15 of 21
(r*s)(4)-(r(4)*s(4))
--R
--R
--R (15) 0
--R Type: NonNegativeInteger
--E 15

--S 16 of 21
t:=(x:INT):EXPR(INT)+-> (x*x*x)
--R
--R
--R (16) theMap(Closure)
--R Type: (Integer -> Expression Integer)
--E 16

--S 17 of 21
u:=(y:INT):EXPR(INT)+-> (y*y+3)
--R
--R
--R (17) theMap(Closure)
--R Type: (Integer -> Expression Integer)
--E 17

--S 18 of 21
(t/u)(4)-(t(4)/u(4))
Given functions $f$ and $g$, returns the applicable closure

We can construct some simple maps that take a variable $x$ into an equation:

$p := (x:EXPR(INT)):EXPR(INT) +-> 3 * x$
$q := (x:EXPR(INT)):EXPR(INT) +-> 2 * x + 3$

Now we can do the four arithmetic operations, $+$, $-$, $*$, / on these newly constructed mappings. Since the maps are from the domain
Expression Integer to the same domain we can also use symbolic values for the argument. All of the following will return 0, showing that function composition is equivalent to the result of doing the operations individually.

\[(p+q)(4)-(p(4)+q(4))\]
\[(p+q)(x)-(p(x)+q(x))\]

\[(p-q)(4)-(p(4)-q(4))\]
\[(p-q)(x)-(p(x)-q(x))\]

\[(p*q)(4)-(p(4)*q(4))\]
\[(p*q)(x)-(p(x)*q(x))\]

\[(p/q)(4)-(p(4)/q(4))\]
\[(p/q)(x)-(p(x)/q(x))\]

We can construct simple maps from Integer to Integer but this limits our ability to do division.

\[
r:=(x:INT):INT\rightarrow (x*x*x)\]
\[
s:=(y:INT):INT\rightarrow (y*y+3)\]

Again, all of these will return 0:

\[(r+s)(4)-(r(4)+s(4))\]
\[(r-s)(4)-(r(4)-s(4))\]
\[(r*s)(4)-(r(4)*s(4))\]

If we want to do division with Integer inputs we create the appropriate map:

\[
t:=(x:INT):EXPR(INT)\rightarrow (x*x*x)\]
\[
u:=(y:INT):EXPR(INT)\rightarrow (y*y+3)\]

\[(t/u)(4)-(t(4)/u(4))\]

We can even recover the original functions if we make a map that always returns the constant 1:

\[
h:=(x:EXPR(INT)):EXPR(INT)\rightarrow 1\]

\[
\text{theMap(Closure)}
\]

\[
\text{Type: (Expression Integer \rightarrow Expression Integer)}
\]

\[(p/h)(x)\]

\[3x\]

\[(q/h)(x)\]

\[3x\]

\[\text{Type: Expression Integer}\]
2x + 3

Type: Expression Integer

See Also:
- )show MappingPackage1
- )help MappingPackage2
- )help MappingPackage3
- )help MappingPackage4

---

MappingPackage4 (MAPPKG4)

Exports:
?*? ?+? ?-? ?/?

--- package MAPPKG4 MappingPackage4 ---

)abbrev package MAPKG4 MappingPackage4
++ Author: Timothy Daly
++ Description:
++ Functional Composition.
++ Given functions f and g, returns the applicable closure

MappingPackage4(A:SetCategory, B:Ring):
with

"+": (A->B, A->B) -> (A->B)
++ \spad(+) does functional addition
++
++X f:=(x:INT):INT +-> 3*x
++X g:=(x:INT):INT +-> 2*x+3
++X (f+g)(4)
"-": (A->B, A->B) -> (A->B)
  ++ \spad{(+) does functional addition
  ++
  +++X f:=(x:INT):INT +-> 3*x
  +++X g:=(x:INT):INT +-> 2*x+3
  +++X (f-g)(4)
"*": (A->B, A->B) -> (A->B)
  ++ \spad{(+) does functional addition
  ++
  +++X f:=(x:INT):INT +-> 3*x
  +++X g:=(x:INT):INT +-> 2*x+3
  +++X (f*g)(4)
"/": (A->Expression(Integer), A->Expression(Integer)) -> (A->Expression(Integer))
  ++ \spad{(+) does functional addition
  ++
  +++X p:=(x:EXPR(INT)):EXPR(INT)+->3*x
  +++X q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
  +++X (p/q)(4)
  +++X (p/q)(x)

== add
  fab ==> (A -> B)
  faei ==> (A -> Expression(Integer))

  funcAdd(g:fab,h:fab,x:A):B == ((g x) + (h x))$B
(a:fab)+(b:fab) == c +-> funcAdd(a,b,c)

  funcSub(g:fab,h:fab,x:A):B == ((g x) - (h x))$B
(a:fab)-(b:fab) == c +-> funcSub(a,b,c)

  funcMul(g:fab,h:fab,x:A):B == ((g x) * (h x))$B
(a:fab)*(b:fab) == c +-> funcMul(a,b,c)

  funcDiv(g:faei,h:faei,x:A):Expression(Integer)
    == ((g x) / (h x))$Expression(Integer)
(a:faei)/(b:faei) == c +-> funcDiv(a,b,c)
package MATCAT2 MatrixCategoryFunctions2

---

MatrixCategoryFunctions2 provides functions between two matrix domains. The functions provided are map and reduce.

See Also:
- )show MatrixCategoryFunctions2
MatrixCategoryFunctions2 (MATCAT2)

Exports:
  map   reduce

— package MATCAT2 MatrixCategoryFunctions2 —

)abbrev package MATCAT2 MatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 21 March 1994
++ Description:
++ \spadtype{MatrixCategoryFunctions2} provides functions between two matrix
++ domains. The functions provided are \spadfun{map} and \spadfun{reduce}.

MatrixCategoryFunctions2(R1,Row1,Col1,M1,R2,Row2,Col2,M2):
 Exports == Implementation where
  R1 : Ring
  Row1 : FiniteLinearAggregate R1
  Col1 : FiniteLinearAggregate R1
  M1 : MatrixCategory(R1,Row1,Col1)
  R2 : Ring
  Row2 : FiniteLinearAggregate R2
  Col2 : FiniteLinearAggregate R2
  M2 : MatrixCategory(R2,Row2,Col2)

Exports ==> with
  map: (R1 -> R2,M1) -> M2
  ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
  map: (R1 -> Union(R2,"failed"),M1) -> Union(M2,"failed")
  ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
  reduce: ((R1,R2) -> R2,M1,R2) -> R2
  ++ \spad{reduce(f,m,r)} returns a matrix n where
  ++ \spad{n[i,j] = f(m[i,j],r)} for all indices i and j.

Implementation ==> add
  minr ==> minRowIndex
maxr => maxRowIndex
minc => minColIndex
maxc => maxColIndex

map(f: (R1 -> R2), m : M1) : M2 ==
  ans : M2 := new(nrows m, ncols m, 0)
  for i in minr(m) .. maxr(m) for k in minr(ans) .. maxr(ans) repeat
    for j in minc(m) .. maxc(m) for l in minc(ans) .. maxc(ans) repeat
      qsetelt_!(ans, k, l, f qelt(m, i, j))
    ans

map(f: (R1 -> (Union(R2, "failed"))), m : M1) : Union(M2, "failed") ==
  ans : M2 := new(nrows m, ncols m, 0)
  for i in minr(m) .. maxr(m) for k in minr(ans) .. maxr(ans) repeat
    for j in minc(m) .. maxc(m) for l in minc(ans) .. maxc(ans) repeat
      (r := f qelt(m, i, j)) = "failed" => return "failed"
      qsetelt_!(ans, k, l, r::R2)
    ans

reduce(f, m, ident) ==
  s := ident
  for i in minr(m) .. maxr(m) repeat
    for j in minc(m) .. maxc(m) repeat
      s := f(qelt(m, i, j), s)
  s

package MCDEN MatrixCommonDenominator

  — MatrixCommonDenominator.input —

)set break resume
)sys rm -f MatrixCommonDenominator.output
)spool MatrixCommonDenominator.output
)set message test on
---

<table>
<thead>
<tr>
<th>MatrixCommonDenominator.help</th>
</tr>
</thead>
</table>

MatrixCommonDenominator examples
====================================================================

MatrixCommonDenominator provides functions to compute the common denominator of a matrix of elements of the quotient field of an integral domain.

See Also:
o )show MatrixCommonDenominator
---

MatrixCommonDenominator (MCDEN)

Exports:
commonDenominator  clearDenominator  splitDenominator

---

package MCDEN MatrixCommonDenominator ---
MatrixCommonDenominator(R, Q): Exports == Implementation where
R: IntegralDomain
Q: QuotientFieldCategory R

VR ==> Vector R
VQ ==> Vector Q

Exports ==> with
commonDenominator: Matrix Q -> R
++ commonDenominator(q) returns a common denominator d for
++ the elements of q.
clearDenominator: Matrix Q -> Matrix R
++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
++ a common denominator for the elements of q.
splitDenominator: Matrix Q -> Record(num: Matrix R, den: R)
++ splitDenominator(q) returns \spad{[p, d]} such that \spad{q = p/d} and d
++ is a common denominator for the elements of q.

Implementation ==> add
import ListFunctions2(Q, R)
import MatrixCategoryFunctions2(Q, VQ, VQ, Matrix Q, R, VR, VR, Matrix R)

clearDenominator m ==
d := commonDenominator m
map(x -> numer(d*x), m)

splitDenominator m ==
d := commonDenominator m
[map(x -> numer(d*x), m), d]

if R has GcdDomain then
  commonDenominator m == lcm map(denom, parts m)
else
  commonDenominator m == reduce("\*", map(denom, parts m), 1)$List(R)

—— MCDEN.dotabb ——
package MATLIN MatrixLinearAlgebraFunctions

— MatrixLinearAlgebraFunctions.input —

)set break resume
)sys rm -f MatrixLinearAlgebraFunctions.output
)spool MatrixLinearAlgebraFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MatrixLinearAlgebraFunctions
--E 1

)spool
)lisp (bye)

— MatrixLinearAlgebraFunctions.help —

====================================================================
MatrixLinearAlgebraFunctions examples
====================================================================

MatrixLinearAlgebraFunctions provides functions to compute inverses and canonical forms.

See Also:
o )show MatrixLinearAlgebraFunctions

———
MatrixLinearAlgebraFunctions (MATLIN)

Exports:
adjoint determinant elColumn2! elRow1! elRow2!
fractionFreeGauss! inverse invertIfCan minordet normalizedDivide
nullSpace nullity rank rowEchelon

— package MATLIN MatrixLinearAlgebraFunctions —

)abbrev package MATLIN MatrixLinearAlgebraFunctions
++ Author: Clifton J. Williamson, P. Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: December 1992
++ Description:
++ \spadtype{MatrixLinearAlgebraFunctions} provides functions to compute
++ inverses and canonical forms.

MatrixLinearAlgebraFunctions(R,Row,Col,M):Exports == Implementation where
R : CommutativeRing
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)
I ==> Integer

Exports ==> with
determinant: M -> R
++ \spad{determinant(m)} returns the determinant of the matrix m.
++ an error message is returned if the matrix is not square.
minordet: M -> R
++ \spad{minordet(m)} computes the determinant of the matrix m using
++ minors. Error: if the matrix is not square.
elRow1! : (M,I,I) -> M
++ elRow1!(m,i,j) swaps rows i and j of matrix m: elementary operation
++ of first kind
elRow2! : (M,R,I,I) -> M
++ elRow2!(m,a,i,j) adds to row i a*row(m,j): elementary operation of
++ second kind. (i ^= j)

\texttt{elColumn2!} : (M, R, I, I) -> M
++ \texttt{elColumn2!(m, a, i, j)} adds to column i a*column(m, j) : elementary
++ operation of second kind. (i ^= j)

if \( R \) has IntegralDomain then
  \texttt{rank: M -> NonNegativeInteger}
  ++ \texttt{\spad{rank(m)}} returns the rank of the matrix \( m \).
  \texttt{nullity: M -> NonNegativeInteger}
  ++ \texttt{\spad{nullity(m)}} returns the nullity of the matrix \( m \). This is
  ++ the dimension of the null space of the matrix \( m \).
  \texttt{nullSpace: M -> List Col}
  ++ \texttt{\spad{nullSpace(m)}} returns a basis for the null space of the
  ++ matrix \( m \).
  \texttt{fractionFreeGauss!: M -> M}
  ++ \texttt{\spad{fractionFreeGauss(m)}} performs the fraction
  ++ free gaussian elimination on the matrix \( m \).
  \texttt{invertIfCan: M -> Union(M, "failed")}
  ++ \texttt{\spad{invertIfCan(m)}} returns the inverse of \( m \) over \( R \)
  \texttt{adjoint: M -> Record(adjMat: M, detMat: R)}
  ++ \texttt{\spad{adjoint(m)}} returns the adjoint matrix of \( m \) (i.e. the matrix
  ++ \( n \) such that \( m*n = \text{determinant}(m)*\text{id} \) and the determinant of \( m \).

if \( R \) has EuclideanDomain then
  \texttt{rowEchelon: M -> M}
  ++ \texttt{\spad{rowEchelon(m)}} returns the row echelon form of the matrix \( m \).
  \texttt{normalizedDivide: \( (R, R) \) -> Record(quotient: R, remainder: R)}
  ++ \texttt{normalizedDivide(n, d)} returns a normalized quotient and
  ++ remainder such that consistently unique representatives
  ++ for the residue class are chosen, e.g. positive remainders

if \( R \) has Field then
  \texttt{inverse: M -> Union(M, "failed")}
  ++ \texttt{\spad{inverse(m)}} returns the inverse of the matrix.
  ++ If the matrix is not invertible, "failed" is returned.
  ++ Error: if the matrix is not square.

Implementation ==> add

\texttt{rowAllZeroes?: (M, I) -> Boolean}
\texttt{rowAllZeroes?(x, i) ==}
-- determines if the \( i \)th row of \( x \) consists only of zeroes
-- internal function: no check on index \( i \)
for \( j \) in \texttt{minColIndex(x)}..\texttt{maxColIndex(x)} repeat
  \texttt{qelt(x, i, j) ^= 0 => return false}
\texttt{true}

\texttt{colAllZeroes?: (M, I) -> Boolean}
\texttt{colAllZeroes?(x, j) ==}
CHAPTER 14. CHAPTER M

-- determines if the ith column of x consists only of zeroes
-- internal function: no check on index j
for i in minRowIndex(x)..maxRowIndex(x) repeat
  qelt(x,i,j) ^= 0 => return false
true

minorDet:(M,I,List I,I,PrimitiveArray(Union(R,"uncomputed")))-> R
minorDet(x,m,l,i,v) ==
  z := v.m
  z case R => z
  ans : R := 0; rl : List I := nil()
  j := first l; l := rest l; pos := true
  minR := minRowIndex x; minC := minColIndex x;
  repeat
    if qelt(x,j + minR,i + minC) ^= 0 then
      ans :=
      md := minorDet(x,m - 2**(j :: NonNegativeInteger),_
        concat_!(reverse rl,l),i + 1,v) *_
        qelt(x,j + minR,i + minC)
      pos => ans + md
      ans - md
    null l =>
      v.m := ans
      return ans
    pos := not pos; rl := cons(j,rl); j := first l; l := rest l

minordet x ==
  (ndim := nrows x) ^= (ncols x) =>
  error "determinant: matrix must be square"
  -- minor expansion with (s---loads of) memory
  n1 : I := ndim - 1
  v : PrimitiveArray(Union(R,"uncomputed")) :=
    new((2**ndim - 1) :: NonNegativeInteger,"uncomputed")
  minR := minRowIndex x; maxC := maxColIndex x
  for i in 0..n1 repeat
    qsetelt_!(v,(2**i - 1),qelt(x,i + minR,maxC))
    minorDet(x, 2**ndim - 2, [i for i in 0..n1], 0, v)

  -- elementary operation of first kind: exchange two rows --
elRow1!(m:M,i:I,j:I) : M ==
  vec:=row(m,i)
  setRow!(m,i,row(m,j))
  setRow!(m,j,vec)
  m

  -- elementary operation of second kind: add to row i--
  -- a*row j (i^=j) --
elRow2!(m : M,a:R,i:I,j:I) : M ==
  vec:= map((r1:R):R +-> a*r1,row(m,j))
  vec:=map("+",row(m,i),vec)
setRow!(m,i,vec)
m
   -- elementary operation of second kind: add to column i --
   -- a*column j (i\neq j) --

elColumn2!(m : M,a:R,i:I,j:I) : M ==
vec:= map((r1:R):R +-> a*r1,column(m,j))
vec:=map("+",column(m,i),vec)
setColumn!(m,i,vec)
m

if R has IntegralDomain then
   -- Fraction-Free Gaussian Elimination
fractionFreeGauss! x ==
   (ndim := nrows x) = 1 => x
   ans := b := 1$R
   minR := minRowIndex x; maxR := maxRowIndex x
   minC := minColIndex x; maxC := maxColIndex x
   i := minR
   for j in minC..maxC repeat
      if qelt(x,i,j) = 0 then -- candidate for pivot = 0
         rown := minR - 1
         for k in (i+1)..maxR repeat
            if qelt(x,k,j) ^= 0 then
               rown := k -- found a pivot
               leave
         if rown > minR - 1 then
            swapRows_!(x,i,rown)
            ans := -ans
      (c := qelt(x,i,j)) = 0 => "next j" -- try next column
      for k in (i+1)..maxR repeat
         if qelt(x,k,j) = 0 then
            for l in (j+1)..maxC repeat
               qsetelt_!(x,k,l,(c * qelt(x,k,l) exquo b) :: R)
               else
                  pv := qelt(x,k,j)
                  qsetelt_!(x,k,j,0)
                  for l in (j+1)..maxC repeat
                     val := c * qelt(x,k,l) - pv * qelt(x,i,l)
                     qsetelt_!(x,k,l,(val exquo b) :: R)
            b := c
            (i := i+1)>maxR => leave
      if ans=-1 then
         lasti := i-1
         for j in 1..maxC repeat x(lasti,j) := -x(lasti,j)
x
   --
lastStep(x:M) : M ==
   ndim := nrows x
   minR := minRowIndex x; maxR := maxRowIndex x
\[
\begin{align*}
\minC & := \minColIndex x; \ \maxC := \minC + \text{ndim} - 1 \\
exCol & := \maxColIndex x \\
det & := x(\maxR, \maxC) \\
\maxR1 & := \maxR - 1 \\
\maxC1 & := \maxC + 1 \\
\minC1 & := \minC + 1 \\
iRow & := \maxR \\
iCol & := \maxC - 1 \\
\text{for } i \text{ in } \maxR1 \ldots 1 \text{ by } -1 \text{ repeat} \\
\text{for } j \text{ in } \maxC1 \ldots \text{exCol} \text{ repeat} \\
ss & := +/[x(i, iCol + k) * x(i + k, j) \text{ for } k \text{ in } 1 .. (\maxR - i)] \\
x(i, j) & := \_\text{exquo}((\det * x(i, j) - ss), x(i, iCol))::R \\
iCol & := iCol - 1 \\
\text{subMatrix}(x, \minR, \maxR, \maxC1, \text{exCol})
\end{align*}
\]

\text{invertIfCan}(y) ==
\[
(\text{nr} := \text{nrows } y) = (\text{ncols } y) \Rightarrow \\
\text{error } "\text{invertIfCan: matrix must be square}"
\]

\text{adjRec} := \text{adjoint } y \\
(\text{den} := \text{recip(adjRec.detMat)}) \text{ case } "\text{failed}" \Rightarrow "\text{failed}"
\[
\text{den} := R * \text{adjRec.adjMat}
\]

\text{adjoint}(y) ==
\[
(\text{nr} := \text{nrows } y) = (\text{ncols } y) \Rightarrow \text{error } "\text{adjoint: matrix must be square}"
\]

\maxR := \text{maxRowIndex } y \\
\maxC := \text{maxColIndex } y \\
x := \text{horizConcat(copy } y, \text{scalarMatrix}(\text{nr}, 1$R)) \\
\text{ffr} := \text{fractionFreeGauss!}(x) \\
\text{det} := \text{ffr(}\maxR, \maxC) \\
[\text{lastStep(ffr)}, \text{det}]
\]

\text{if } R \text{ has Field then}
\[
\begin{align*}
\text{VR} & \quad \Rightarrow \text{Vector } R \\
\text{IMATLIN} & \quad \Rightarrow \text{InnerMatrixLinearAlgebraFunctions}(R, \text{Row}, \text{Col}, M) \\
\text{MMATLIN} & \quad \Rightarrow \text{InnerMatrixLinearAlgebraFunctions}(R, \text{VR}, \text{VR}, \text{Matrix } R) \\
\text{FLA2} & \quad \Rightarrow \text{FiniteLinearAggregateFunctions2}(R, \text{VR}, R, \text{Col}) \\
\text{MAT2} & \quad \Rightarrow \text{MatrixCategoryFunctions2}(R, \text{Row}, \text{Col}, M, R, VR, VR, \text{Matrix } R)
\end{align*}
\]

\text{rowEchelon } y \Rightarrow \text{rowEchelon}(y)$\text{IMATLIN}$ \\
\text{rank } y \Rightarrow \text{rank}(y)$\text{IMATLIN}$ \\
\text{nullity } y \Rightarrow \text{nullity}(y)$\text{IMATLIN}$ \\
\text{determinant } y \Rightarrow \text{determinant}(y)$\text{IMATLIN}$ \\
\text{inverse } y \Rightarrow \text{inverse}(y)$\text{IMATLIN}$ \\
\text{if } \text{Col has shallowlyMutable then} \\
\text{nullSpace } y \Rightarrow \text{nullSpace}(y)$\text{IMATLIN}$ \\
\text{else} \\
\text{nullSpace } y \Rightarrow \\
[\text{map}((r1:R):R \leftrightarrow r1, v)$\text{FLA2}$]
for v in nullSpace(map((r2:R):R +-> r2, y)$MAT2)$MMATLIN]

else if R has IntegralDomain then
  QF ==> Fraction R
  Row2 ==> Vector QF
  Co2 ==> Vector QF
  M2 ==> Matrix QF
  IMATQF ==> InnerMatrixQuotientFieldFunctions(R,Row,Co,M,QF,Row2,Co2,M2)

nullSpace m == nullSpace(m)$IMATQF

determinant y ==
  (nrows y) ^= (ncols y) => error "determinant: matrix must be square"
  fm:=fractionFreeGauss!(copy y)
  fm(maxRowIndex fm,maxColIndex fm)

rank x ==
  y :=
  (rk := nrows x) > (rh := ncols x) =>
    rk := rh
    transpose x
    copy x
  y := fractionFreeGauss! y
  i := maxRowIndex y
  while rk > 0 and rowAllZeroes?(y,i) repeat
    i := i - 1
    rk := (rk - 1) :: NonNegativeInteger
  rk :: NonNegativeInteger

nullity x == (ncols x - rank x) :: NonNegativeInteger

if R has EuclideanDomain then
  if R has IntegerNumberSystem then
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      qr := divide(n, d)
      qr.remainder >= 0 => qr
      d > 0 =>
        qr.remainder := qr.remainder + d
        qr.quotient := qr.quotient - 1
        qr
        qr.remainder := qr.remainder - d
        qr.quotient := qr.quotient + 1
        qr
  else
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      divide(n, d)

rowEchelon y ==
  x := copy y
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
n := minR - 1
i := minR
for j in minC..maxC repeat
  if i > maxR then leave x
  n := minR - 1
  xnj := R
  for k in i..maxR repeat
    if not zero?(xkj:=qelt(x,k,j)) and ((n = minR - 1) or sizeLess?(xkj,xnj)) then
      n := k
      xnj := xkj
      n = minR - 1 => "next j"
    swapRows_!(x,i,n)
  end
  for k in (i+1)..maxR repeat
    qelt(x,k,j) = 0 => "next k"
  end
  aa := extendedEuclidean(qelt(x,i,j),qelt(x,k,j))
  (a,b,d) := (aa.coef1,aa.coef2,aa.generator)
  b1 := (qelt(x,i,j) exquo d) :: R
  a1 := (qelt(x,k,j) exquo d) :: R
  a*b1+a1*b = 1
  for k1 in (j+1)..maxC repeat
    val1 := a * qelt(x,i,k1) + b * qelt(x,k,k1)
    val2 := -a1 * qelt(x,i,k1) + b1 * qelt(x,k,k1)
    qsetelt_!(x,i,k1,val1); qsetelt_!(x,k,k1,val2)
    qsetelt_!(x,i,j,d); qsetelt_!(x,k,j,0)
  end
  un := unitNormal qelt(x,i,j)
  qsetelt_!(x,i,j,un.canonical)
  if un.associate ^= 1 then for jj in (j+1)..maxC repeat
    qsetelt_!(x,i,jj,un.associate * qelt(x,i,jj))
  end
  xij := qelt(x,i,j)
  for k in minR..(i-1) repeat
    qelt(x,k,j) = 0 => "next k"
    qr := normalizedDivide(qelt(x,k,j), xij)
    qsetelt_!(x,k,j,qr.remainder)
    for k1 in (j+1)..maxC repeat
      qsetelt_!(x,k,k1,qelt(x,k,k1) - qr.quotient * qelt(x,i,k1))
  end
  i := i + 1
  x
else
determinant x == minordet x

— MATLIN.dotabb —
package MAMA MatrixManipulation

--- MatrixManipulation.input ---

)set break resume
)sys rm -f MatrixManipulation.output
)spool MatrixManipulation.output
)set message test on
)set message auto off
)clear all

--S 1 of 44
M := matrix([[a,b,c],[d,e,f],[g,h,i]])
--R
--R
--R +a b c+
--R | |   
--R (1) |d e f|
--R | |   
--R +g h i+   Type: Matrix(Polynomial(Integer))
--E 1

--S 2 of 44
element(M, 2,2)
--R
--R
--R (2) [e]   Type: Matrix(Fraction(Polynomial(Integer)))
--E 2

--S 3 of 44
aRow(M, 1)
--R
--R
--R (3) [a b c]   Type: Matrix(Fraction(Polynomial(Integer)))
--E 3

--S 4 of 44
aRow(M, 2)
--R
--R
--R (4) [d e f]
    Type: Matrix(Fraction(Polynomial(Integer)))
--E 4

--S 5 of 44
aColumn(M, 2)
--R
--R
    +b+
    Type: Matrix(Fraction(Polynomial(Integer)))
--E 5

--S 6 of 44
aColumn(M, 3)
--R
--R
    +c+
    Type: Matrix(Fraction(Polynomial(Integer)))
--E 6

--S 7 of 44
rows(M, [1,2])
--R
--R
    +a b c+
    Type: Matrix(Fraction(Polynomial(Integer)))
--E 7

--S 8 of 44
rows(M, [1,3])
--R
--R
    +a b c+
    Type: Matrix(Fraction(Polynomial(Integer)))
--E 8
--S 9 of 44
rows(M, [3,2])
--R
--R
--R  +g h i+
--R  (9) |   |
--R  +d e f+
--R
Type: Matrix(Fraction(Polynomial(Integer)))
--E 9

--S 10 of 44
rows(M, 2..3)
--R
--R
--R  +d e f+
--R  (10) |   |
--R  +g h i+
--R
Type: Matrix(Fraction(Polynomial(Integer)))
--E 10

--S 11 of 44
columns(M, [1,2])
--R
--R
--R  +a b+
--R  |   |
--R  (11) |d e|
--R  |   |
--R  +g h+
--R
Type: Matrix(Fraction(Polynomial(Integer)))
--E 11

--S 12 of 44
columns(M, [1,3])
--R
--R
--R  +a c+
--R  |   |
--R  (12) |d f|
--R  |   |
--R  +g i+
--R
Type: Matrix(Fraction(Polynomial(Integer)))
--E 12

--S 13 of 44
columns(M, [3,2])
--R
--R
--R  +c b+
Type: Matrix(Fraction(Polynomial(Integer)))

Type: Matrix(Fraction(Polynomial(Integer)))

Type: Matrix(Fraction(Polynomial(Integer)))

Type: Matrix(Fraction(Polynomial(Integer)))

Type: Matrix(Fraction(Polynomial(Integer)))

Type: Matrix(Fraction(Polynomial(Integer)))
diagonalMatrix(M, 1)

\[
\begin{pmatrix}
+0 & b & 0 \\
0 & 0 & f \\
0 & 0 & 0
\end{pmatrix}
\]

Type: Matrix(Fraction(Polynomial(Integer)))

---E 18

---S 19 of 44
diagonalMatrix(M, 2)

\[
\begin{pmatrix}
+0 & 0 & c \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

Type: Matrix(Fraction(Polynomial(Integer)))

---E 19

---S 20 of 44
diagonalMatrix(M, -1)

\[
\begin{pmatrix}
+0 & 0 & 0 \\
d & 0 & 0 \\
0 & h & 0
\end{pmatrix}
\]

Type: Matrix(Fraction(Polynomial(Integer)))

---E 20

---S 21 of 44
diagonalMatrix(M, -2)

\[
\begin{pmatrix}
+0 & 0 & 0 \\
0 & 0 & 0 \\
g & 0 & 0
\end{pmatrix}
\]

Type: Matrix(Fraction(Polynomial(Integer)))

---E 21

---S 22 of 44
bandMatrix(M, [-1,1])
--R
--R
--R +0 b 0+
--R | |
--R (22) |d 0 f|
--R | |
--R +0 h 0+
--R

Type: Matrix(Fraction(Polynomial(Integer)))
--E 22

--S 23 of 44
bandMatrix(M, [-1,0,1])
--R
--R
--R +a b 0+
--R | |
--R (23) |d e f|
--R | |
--R +0 h i+
--R

Type: Matrix(Fraction(Polynomial(Integer)))
--E 23

--S 24 of 44
bandMatrix(M, -1..1)
--R
--R
--R +a b 0+
--R | |
--R (24) |d e f|
--R | |
--R +0 h i+
--R

Type: Matrix(Fraction(Polynomial(Integer)))
--E 24

-- Build a larger matrix

--S 25 of 44
A := matrix([[a]])
--R
--R
--R (25) [a]
--R

Type: Matrix(Polynomial(Integer))
--E 25

--S 26 of 44
B := matrix([[b]])
--R
--R
--R (26) [b]
C := matrix([[c]])

A11 := element(M, 3,3)

A12 := horizConcat([A,B,C])

A21 := vertConcat([A,B,C])

E := blockConcat([[A11,A12],[A21,M]])
--R Type: Matrix(Fraction(Polynomial(Integer)))
--E 31

-- Split again

--S 32 of 44
vertSplit(E, 2)
--R
--R
--R +i a b c+ +b d e f+
--R (32) [[|  |  |  |]]
--R +a a b c+ +c g h i+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 32

--S 33 of 44
horizSplit(E, 2)
--R
--R
--R +i a+ +b c+
--R | | | |
--R |a a| |b c|
--R (33) [[|  |  |  |]]
--R |b d| |e f|
--R | | | |
--R +c g+ +h i+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 33

--S 34 of 44
vertSplit(E, [1,2,1])
--R
--R
--R +a a b c+
--R (34) [[i a b c],[|,|,|,c g h i]]
--R +b d e f+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 34

--S 35 of 44
horizSplit(E, [2,2])
--R
--R
--R +i a+ +b c+
--R | | | |
--R |a a| |b c|
--R (35) [[|  |  |  |]]
--R |b d| |e f|
--R | | | |
--R +c g+ +h i+
--R  Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 35

--S 36 of 44
blockSplit(E, 2,2)
--R
--R
--R +i a+ +b c+ +b d+ +e f+
--R (36) [[],[|],|],[|],[|],[|]]
--R +a a+ +b c+ +c g+ +h i+
--R  Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
--E 36

--S 37 of 44
zero?(blockConcat(%) - E)
--R
--R
--R (37) true
--R  Type: Boolean
--E 37

--S 38 of 44
blockSplit(E, [1,2,1], [2,2])
--R
--R
--R +a a+ +b c+
--R (38) [[[i a],[b c]],[|],[|],[|],[c g],[h i]]]
--R +b d+ +e f+
--R  Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
--E 38

--S 39 of 44
zero?(blockConcat(%) - E)
--R
--R
--R (39) true
--R  Type: Boolean
--E 39

--S 40 of 44
blockSplit(E, [2,1,1], 2)
--R
--R
--R +i a+ +b c+
--R (40) [[],[|],[|],[b d],[e f],[c g],[h i]]]
--R +a a+ +b c+
--R  Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
--E 40

--S 41 of 44
CHAPTER 14. CHAPTER M

zero?(blockConcat(%) - E)
--R
--R
--R (41) true
--R
--E 41

--S 42 of 44
blockSplit(E, 4, [2,2])
--R
--R
--R (42) [[[i a],[b c]],[[a a],[b c]],[[b d],[e f]],[[c g],[h i]]]
--R
Type: List(List(Matrix(Fraction(Polynomial(Integer)))))
--E 42

--S 43 of 44
zero?(blockConcat(%) - E)
--R
--R
--R (43) true
--R
--E 43

--S 44 of 44
)show MatrixManipulation
--E 44

)spool
)lisp (bye)

— MatrixManipulation.help —

====================================================================
MatrixManipulation examples
====================================================================

Some functions for manipulating (dense) matrices.

Supported are various kinds of slicing, splitting and stacking of matrices. The functions resemble operations often used in numerical linear algebra algorithms.

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
+a b c+
\[
\begin{array}{ccc}
| & | \\
|d e f| \\
| | \\
+g h i+
\end{array}
\]

\text{element}(M, 2, 2)
\[[e]\]

\text{aRow}(M, 1)
\[[a b c]\]

\text{aRow}(M, 2)
\[[d e f]\]

\text{aColumn}(M, 2)
\n
\n
\text{aColumn}(M, 3)
\n
\n
\text{rows}(M, [1,2])
\n
\n
\text{rows}(M, [1,3])
\n
\n
\text{rows}(M, [3,2])
\n
\n
rows(M, 2..3)
+  d  e  f +
|   |   |
+  g  h  i +

columns(M, [1,2])
+  a  b +
|   |   |
|d  e|
|   |   |
+  g  h +

columns(M, [1,3])
+  a  c +
|   |   |
|d  f|
|   |   |
+  g  i +

columns(M, [3,2])
+  c  b +
|   |   |
|f  e|
|   |   |
+i  h +

columns(M, 1..2)
+  a  b +
|   |   |
|d  e|
|   |   |
+  g  h +

subMatrix(M, [1,2],[1,2])
+  a  b +
|   |   |
+  d  e +

subMatrix(M, [1,3],[1,3])
+  a  c +
|   |   |
+  g  i +
diagonalMatrix(M)

\[
\begin{bmatrix}
+a & 0 & 0 \\
0 & e & 0 \\
0 & 0 & i \\
\end{bmatrix}
\]

\[
\text{diagonalMatrix}(M, 1)
\]

\[
\begin{bmatrix}
+0 & b & 0 \\
0 & f & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{diagonalMatrix}(M, 2)
\]

\[
\begin{bmatrix}
+0 & 0 & c \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{diagonalMatrix}(M, -1)
\]

\[
\begin{bmatrix}
+0 & 0 & 0 \\
0 & d & 0 \\
0 & h & 0 \\
\end{bmatrix}
\]

\[
\text{diagonalMatrix}(M, -2)
\]

\[
\begin{bmatrix}
+0 & 0 & 0 \\
0 & g & 0 \\
\end{bmatrix}
\]

\[
\text{bandMatrix}(M, [-1,1])
\]

\[
\begin{bmatrix}
+0 & b & 0 \\
0 & d & f \\
0 & h & 0 \\
\end{bmatrix}
\]

\[
\text{bandMatrix}(M, [-1,0,1])
\]
Build a larger matrix

\[ A := \text{matrix}([a]) \]

\[ A_{11} := \text{element}(M, 3, 3) \]

\[ A_{12} := \text{horizConcat}([A, B, C]) \]

\[ A_{21} := \text{vertConcat}([A, B, C]) \]

\[ E := \text{blockConcat}([[A_{11}, A_{12}], [A_{21}, M]]) \]
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b d e f</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>c g h i</td>
<td></td>
</tr>
</tbody>
</table>

Split again

vertSplit(E, 2)

+i a b c+ +b d e f+
[[ | |,| |]]
+a a b c+ +c g h i+

horizSplit(E, 2)

+i a+ +b c+
| | | |
|a a| b c|
[[ | |,| |]]
|b d| e f|
| | | |
+c g+ +h i+

vertSplit(E, [1,2,1])

+a a b c+
[[i a b c], | |,[c g h i]]
+b d e f+

horizSplit(E, [2,2])

+i a+ +b c+
| | | |
|a a| b c|
[[ | |,| |]]
|b d| e f|
| | | |
+c g+ +h i+

blockSplit(E, 2,2)

+i a+ +b c+ +b d+ +e f+
[[| |,| |],| |,[| |,| |]]
+a a+ +b c+ +c g+ +h i+

zero?(blockConcat(%) - E)

true

blockSplit(E, [1,2,1], [2,2])
CHAPTER M

\[
\begin{bmatrix}
  a & a+ b & c +  \\
  \begin{bmatrix}
    a & c \\
    c & g
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  d & e & f +  \\
  \begin{bmatrix}
    b & d \\
    e & f
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g & h & i +  \\
  \begin{bmatrix}
    c & g \\
    h & i
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a & a & b & c +  \\
  \begin{bmatrix}
    a & a \\
    b & c
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  d & e & f +  \\
  \begin{bmatrix}
    d & e \\
    f & g
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g & h & i +  \\
  \begin{bmatrix}
    g & h \\
    i & j
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a & a & b & c +  \\
  \begin{bmatrix}
    a & a \\
    b & c
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  d & e & f +  \\
  \begin{bmatrix}
    d & e \\
    f & g
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g & h & i +  \\
  \begin{bmatrix}
    g & h \\
    i & j
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a & a & b & c +  \\
  \begin{bmatrix}
    a & a \\
    b & c
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  d & e & f +  \\
  \begin{bmatrix}
    d & e \\
    f & g
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g & h & i +  \\
  \begin{bmatrix}
    g & h \\
    i & j
  \end{bmatrix}
\end{bmatrix}
\]

See Also:

- show MatrixManipulation

---

MatrixManipulation (MAMA)
Exports:
apColumn aRow bandMatrix blockConcat blockSplit
columns diagonalMatrix element horizConcat horizSplit
rows subMatrix vertConcat vertSplit

--- package MAMA MatrixManipulation ---

)abbrev package MAMA MatrixManipulation
++ Author: Raoul Bourquin
++ Date Created: 17 November 2012
++ Date Last Updated: 1 December 2012
++ Description:
++ Some functions for manipulating (dense) matrices.
++ Supported are various kinds of slicing, splitting and stacking of
++ matrices. The functions resemble operations often used in numerical
++ linear algebra algorithms.
MatrixManipulation(R, Row, Col, M) : Exports == Implementation where
  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

I ==> Integer
PI ==> PositiveInteger
LI ==> List I
SI ==> Segment I
LPI ==> List PI
SPI ==> Segment PI

Exports == with

--- Slicing matrices

-- How to call aRow, aColumn? Name clashed with usual row, column
-- Package call is ugly because of many parameters of MAMA

element : (M, PI, PI) -> M
  ++ \spad{element} returns a single element out of a matrix.
  ++ The element is put into a one by one matrix.

aRow : (M, PI) -> M
  ++ \spad{aRow} returns a single row out of a matrix.
  ++ The row is put into a one by N matrix.

rows : (M, LPI) -> M
  ++ \spad{rows} returns several rows out of a matrix.
  ++ The rows are stacked into a matrix.

rows : (M, SPI) -> M
  ++ \spad{rows} returns several rows out of a matrix.
++ The rows are stacked into a matrix.

\texttt{aColumn} : (\texttt{M}, \texttt{PI}) -> \texttt{M}
++ \texttt{aColumn} returns a single column out of a matrix.
++ The column is put into a one by \texttt{N} matrix.

\texttt{columns} : (\texttt{M}, \texttt{LPI}) -> \texttt{M}
++ \texttt{columns} returns several columns out of a matrix.
++ The columns are stacked into a matrix.

\texttt{columns} : (\texttt{M}, \texttt{SPI}) -> \texttt{M}
++ \texttt{columns} returns several columns out of a matrix.
++ The columns are stacked into a matrix.

\texttt{subMatrix} : (\texttt{M}, \texttt{LPI}, \texttt{LPI}) -> \texttt{M}
++ \texttt{subMatrix} returns several elements out of a matrix.
++ The elements are stacked into a submatrix.

\texttt{subMatrix} : (\texttt{M}, \texttt{SPI}, \texttt{SPI}) -> \texttt{M}
++ \texttt{subMatrix} returns several elements out of a matrix.
++ The elements are stacked into a submatrix.

\texttt{diagonalMatrix} : (\texttt{M}, \texttt{I}) -> \texttt{M}
++ \texttt{diagonalMatrix} returns a diagonal out of a matrix.
++ The diagonal is put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.

\texttt{diagonalMatrix} : \texttt{M} -> \texttt{M}
++ \texttt{diagonalMatrix} returns the main diagonal out of
++ a matrix. The diagonal is put into a matrix of same shape
++ as the original one.

\texttt{bandMatrix} : (\texttt{M}, \texttt{LI}) -> \texttt{M}
++ \texttt{bandMatrix} returns multiple diagonals out of a matrix.
++ The diagonals are put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.

\texttt{bandMatrix} : (\texttt{M}, \texttt{SI}) -> \texttt{M}
++ \texttt{bandMatrix} returns multiple diagonals out of a matrix.
++ The diagonals are put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.

-- Stacking matrices

\texttt{horizConcat} : (\texttt{List} \texttt{M}) -> \texttt{M}
++ \texttt{horizConcat} concatenates matrices column wise.
vertConcat : (List M) -> M
  ++ \spad{vertConcat} concatenates matrices row wise.

blockConcat : (List List M) -> M
  ++ \spad{blockConcat} concatenates matrices row and
  ++ column wise, building a block matrix. The order
  ++ is row major as in \spad{matrix}.

-- Splitting matrices

vertSplit : (M, PI) -> List M
  ++ \spad{vertSplit} splits a matrix into multiple
  ++ submatrices row wise.

vertSplit : (M, LPI) -> List M
  ++ \spad{vertSplit} splits a matrix into multiple
  ++ submatrices row wise.

horizSplit : (M, PI) -> List M
  ++ \spad{horizSplit} splits a matrix into multiple
  ++ submatrices column wise.

horizSplit : (M, LPI) -> List M
  ++ \spad{horizSplit} splits a matrix into multiple
  ++ submatrices column wise.

blockSplit : (M, PI, PI) -> List List M
  ++ \spad{blockSplit} splits a matrix into multiple
  ++ submatrices row and column wise, dividing
  ++ a matrix into blocks.

blockSplit : (M, LPI, PI) -> List List M
  ++ \spad{blockSplit} splits a matrix into multiple
  ++ submatrices row and column wise, dividing
  ++ a matrix into blocks.

blockSplit : (M, PI, LPI) -> List List M
  ++ \spad{blockSplit} splits a matrix into multiple
  ++ submatrices row and column wise, dividing
  ++ a matrix into blocks.

blockSplit : (M, LPI, LPI) -> List List M
  ++ \spad{blockSplit} splits a matrix into multiple
  ++ submatrices row and column wise, dividing
  ++ a matrix into blocks.

Implementation ==> add

minr => minRowIndex
maxr => maxRowIndex
minc ==> minColIndex
maxc ==> maxColIndex

-- Custom function to expand Segment(PositiveInteger) into
-- List(PositiveInteger). This operation is not supported by the
-- overly restrictive library implementation.
expand(spi : SPI) : LPI ==
lr := empty$LPI
l : PI := lo spi
h : PI := hi spi
inc : I := incr spi
zero? inc => error "Cannot expand a segment with an increment of zero"
if inc > 0 then
  while l <= h repeat
    lr := concat(l, lr)
    l := (l + inc) pretend PI
  else
    while l >= h repeat
      lr := concat(l, lr)
      l := (l + inc) pretend PI
  reverse! lr

element(A, r, c) ==
  matrix([[A(r,c)]])
aRow(A:M, r:PI) : M ==
  subMatrix(A, r, r, minc A, maxc A)
rows(A:M, lst:LPI) : M ==
  ls := [aRow(A, r) for r in lst]
  reduce(vertConcat, ls)
rows(A:M, si:SPI) : M ==
  rows(A, expand(si))
aColumn(A:M, c:PI) : M ==
  subMatrix(A, minr A, maxr A, c, c)
columns(A:M, lst:LPI) : M ==
  ls := [aColumn(A, c) for c in lst]
  reduce(horizConcat, ls)
columns(A:M, si:SPI) : M ==
  columns(A, expand(si))
diagonalMatrix(A, n) ==
  nr := nrows(A)
  nc := ncols(A)
  n > (nc-1) => error "requested diagonal out of range"
  n < 0 and abs(n) > (nr-1) => error "requested diagonal out of range"
\begin{verbatim}
B := zero(nr,nc)
if n >= 0 then
dl := min(nc-n, nr)
sr := minr(A)
sc := minc(A) + n
else
dl := min(nc, nr-abs(n))
sr := minr(A) + abs(n)
sc := minc(A)
for i in 0..(dl-1) repeat
  qsetelt!(B, sr+i, sc+i, A(sr+i, sc+i))
B

diagonalMatrix(A) ==
diagonalMatrix(A, 0)

bandMatrix(A:M, ln:LI) : M ==
  -- Really inefficient
  reduce("+", [diagonalMatrix(A,d) for d in ln])

bandMatrix(A:M, si:SI) : M ==
  bandMatrix(A, expand(si))

subMatrix(A:M, lr:LPI, lc:LPI) : M ==
  -- Really inefficient
  lle := [[ element(A,r,c) for c in lc] for r in lr]
  blockConcat(lle)

subMatrix(A:M, sr:SPI, sc:SPI) : M ==
  subMatrix(A, low sr, high sr, low sc, high sc)

  -- Stack matrices
horizConcat(LA) ==
  reduce(horizConcat, LA)

vertConcat(LA) ==
  reduce(vertConcat, LA)

blockConcat(LLA: List List M) : M ==
  reduce(vertConcat, [reduce(horizConcat, LA) for LA in LLA])

  -- Split matrices
vertSplit(A:M, r:PI) : List M ==
dr := nrows(A) exquo r
dr case "failed" => error "split does not result in an equal division"
  mir := minr A
  mic := minc A
  mac := maxc A
\end{verbatim}
[ subMatrix(A, mir+i*dr, mir+(i+1)*dr-1, mic, mac) for i in 0..(r-1) ]

vertSplit(A:M, lr:LPI) : List M ==
  reduce("+", lr) "= nRows(A) => _
    error "split does not result in proper partition"
  l : List PI := cons(1, scan(_+, lr, 1$PI)$ListFunctions2(PI,PI))
  mir := minr(A) -1 -- additional shift because 1 starts at 1
  mic := minc A
  mac := maxc A
  result := _
    [ subMatrix(A, mir+l(i-1), mir+l(i)-1, mic, mac) for i in 2..#l ]

horizSplit(A:M, c:PI) : List M ==
  dc := ncols(A) exquo c
  dc case "failed" => error "split does not result in an equal division"
  mir := minr A
  mar := maxr A
  mic := minc A
  result := _
    [ subMatrix(A, mir, mar, mic+i*dc, mic+(i+1)*dc-1) for i in 0..(c-1) ]

horizSplit(A:M, lc:LPI) : List M ==
  reduce("+", lc) "= ncols(A) => _
    error "split does not result in proper partition"
  l : List PI := cons(1, scan(_+, lc, 1$PI)$ListFunctions2(PI,PI))
  mir := minr A
  mar := maxr A
  mic := minc(A) -1 -- additional shift because 1 starts at 1
  result := _
    [ subMatrix(A, mir, mar, mic+l(i-1), mic+l(i) for i in 2..#l ]

blockSplit(A:M, nr:PI, nc:PI) : List List M ==
  [ horizSplit(X, nc) for X in vertSplit(A, nr) ]

blockSplit(A:M, lr:LPI, nc:PI) : List List M ==
  [ horizSplit(X, nc) for X in vertSplit(A, lr) ]

blockSplit(A:M, nr:PI, lc:LPI) : List List M ==
  [ horizSplit(X, lc) for X in vertSplit(A, nr) ]

blockSplit(A:M, lr:LPI, lc:LPI) : List List M ==
  [ horizSplit(X, lc) for X in vertSplit(A, lr) ]
"MAMA" -> "MATCAT"

---

package MTHING MergeThing

--- MergeThing.input ---

)set break resume
)sys rm -f MergeThing.output
)spool MergeThing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MergeThing
--E 1

)spool
)lisp (bye)

---

--- MergeThing.help ---

===============================================
MergeThing examples
===============================================

This package exports tools for merging lists

See Also:
 o )show MergeThing

---
CHAPTER 14. CHAPTER M

MergeThing (MTHING)

Exports:
mergeDifference

— package MTHING MergeThing —

)abbrev package MTHING MergeThing
++ Description:
++ This package exports tools for merging lists

MergeThing(S:OrderedSet): Exports == Implementation where
Exports == with
mergeDifference: (List(S),List(S)) -> List(S)
++ mergeDifference(l1,l2) returns a list of elements in l1 not present
++ in l2. Assumes lists are ordered and all x in l2 are also in l1.
Implementation == add
mergeDifference1: (List S,S,List S) -> List S
mergeDifference(x,y) ==
null x or null y => x
mergeDifference1(x,y.first,y.rest)
x.first=y.first => x.rest
x
mergeDifference1(x,fy,ry) ==
rx := x
while not null rx repeat
rx := rx.rest
frx := rx.first
while fy < frx repeat
null ry => return x
fy := first ry
ry := rest ry
frx = fy =>
x.rest := rx.rest
null ry => return x
fy := ry.first
ry := ry.first
x := rx

---

-- MTHING.dotabb --

"MTHING" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MTHING"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"MTHING" -> "FLAGG"

---

package MESH MeshCreationRoutinesForThreeDimensions

--- MeshCreationRoutinesForThreeDimensions.input ---

)set break resume
)sys rm -f MeshCreationRoutinesForThreeDimensions.output
)spool MeshCreationRoutinesForThreeDimensions.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show MeshCreationRoutinesForThreeDimensions
-- E 1

)spool
)lisp (bye)

---

-- MeshCreationRoutinesForThreeDimensions.help ---

====================================================================
MeshCreationRoutinesForThreeDimensions examples
====================================================================

This package has no description

See Also:
  o )show MeshCreationRoutinesForThreeDimensions
MeshCreationRoutinesForThreeDimensions (MESH)

Exports:
meshFun2Var meshPar1Var meshPar2Var ptFunc

— package MESH MeshCreationRoutinesForThreeDimensions —

)abbrev package MESH MeshCreationRoutinesForThreeDimensions
++ Author: Jim Wen
++ Date Last Updated: October 1991 by Jon Steinbach
++ Description:
++ This package has no description

MeshCreationRoutinesForThreeDimensions():Exports == Implementation where

I ==> Integer
PI ==> PositiveInteger
SF ==> DoubleFloat
L ==> List
SEG ==> Segment
S ==> String
Fn1 ==> SF -> SF
Fn2 ==> (SF,SF) -> SF
Fn3 ==> (SF,SF,SF) -> SF
FnPt ==> (SF,SF) -> Point(SF)
FnU ==> Union(Fn3,"undefined")
EX ==> Expression
DROP ==> DrawOption
POINT ==> Point(SF)
SPACE3 ==> ThreeSpace(SF)
COMPPROP ==> SubSpaceComponentProperty
TUBE ==> TubePlot
Exports ==> with
meshPar2Var: (Fn2,Fn2,Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
  ++ meshPar2Var(f,g,h,j,s1,s2,l) undocumented
meshPar2Var: (FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
  ++ meshPar2Var(f,s1,s2,l) undocumented
meshPar2Var: (SPACE3,FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
  ++ meshPar2Var(sp,f,s1,s2,l) undocumented
meshFun2Var: (Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
  ++ meshFun2Var(f,g,s1,s2,l) undocumented
meshPar1Var: (EX I,EX I,EX I,Fn1,SEG SF,L DROP) -> SPACE3
  ++ meshPar1Var(s,t,u,f,s1,l) undocumented
ptFunc: (Fn2,Fn2,Fn2,Fn3) -> ((SF,SF) -> POINT)
  ++ ptFunc(a,b,c,d) is an internal function exported in
  ++ order to compile packages.

Implementation ==> add
import ViewDefaultsPackage()
import SubSpaceComponentProperty()
import DrawOptionFunctions0
import SPACE3
--import TUBE()

-- local functions
numberCheck(nums:Point SF):Void ==
  -- this function checks to see that the small floats are
  -- actually just that - rather than complex numbers or
  -- whatever (the whatever includes nothing presently
  -- since NaN, Not a Number, is not necessarily supported
  -- by common lisp). note that this function is dependent
  -- upon the fact that Common Lisp supports complex numbers.
  for i in minIndex(nums)..maxIndex(nums) repeat
    COMPLEXP(nums.(i::PositiveInteger))$Lisp =>
      error "An unexpected complex number was encountered in the calculations."
makePt:(SF,SF,SF,SF) -> POINT
makePt(x,y,z,c) == point(l : List SF := [x,y,z,c])
ptFunc(f,g,h,c) ==
  (z1:SF,z2:SF):POINT ->
    x := f(z1,z2); y := g(z1,z2); z := h(z1,z2)
    makePt(x,y,z,c(x,y,z))

-- parameterized equations of two variables
meshPar2Var(sp,ptFun,uSeg,vSeg,opts) ==
  -- the issue of open and closed needs to be addressed, here, we are
  -- defaulting to open (which is probably the correct default)
  -- the user should be able to override that (optional argument?)
  l1p : L L POINT := nil()
  uNum : PI := variStepsDefault()
  vNum : PI := variStepsDefault()
ustep := (lo uSeg - hi uSeg)/uNum
vstep := (lo vSeg - hi vSeg)/vNum
someV := hi vSeg
for iv in vNum..0 by -1 repeat
  if zero? iv then someV := lo vSeg
  -- hack: get last number in segment within segment
  lp := L POINT := nil()
someU := hi uSeg
for iu in uNum..0 by -1 repeat
  if zero? iu then someU := lo uSeg
  -- hack: get last number in segment within segment
  pt := ptFun(someU,someV)
  numberCheck pt
  lp := concat(pt,lp)
someU := someU + ustep
llp := concat(lp,llp)
someV := someV + vstep
  -- now llp contains a list of lists of points
  -- for a surface that is a result of a function of 2 variables,
  -- the main component is open and each sublist is open as well
lProp : L COMPPROP := [ new() for l in llp ]
for aProp in lProp repeat
  close(aProp,false)
solid(aProp,false)
aProp : COMPPROP := new()
close(aProp,false)
solid(aProp,false)
space := sp
  -- space := create3Space()
mesh(space,llp,lProp,aProp)
space

meshPar2Var(ptFun,uSeg,vSeg,opts) ==
sp := create3Space()
meshPar2Var(sp,ptFun,uSeg,vSeg,opts)

zCoord: (SF,SF,SF) -> SF
zCoord(x,y,z) == z
meshPar2Var(xFun,yFun,zFun,colorFun,uSeg,vSeg,opts) ==
  -- the color function should be parameterized by (u,v) as well,
  -- not (x,y,z) but we also want some sort of consistency and so
  -- changing this over would mean possibly changing the explicit
  -- stuff over and there, we probably do want the color function
  -- to be parameterized by (x,y,z) - not just (x,y) (this being
  -- for convinence only since z is also defined in terms of (x,y)).
  (colorFun case Fn3) =>
meshPar2Var(ptFunc(xFun,yFun,zFun,colorFun :: Fn3),uSeg,vSeg,opts)
meshPar2Var(ptFunc(xFun,yFun,zFun,zCoord),uSeg,vSeg,opts)
-- explicit equations of two variables
meshFun2Var(zFun,colorFun,xSeg,ySeg,opts) ==
   -- here, we construct the data for a function of two variables
   meshPar2Var((z1:SF,z2:SF):SF +-> z1,
                (x1:SF,x2:SF):SF +-> x2,zFun,colorFun,xSeg,ySeg,opts)

________

— MESH.dotabb —

"MESH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MESH"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"MESH" -> "FIELD"
"MESH" -> "RADCAT"

________

package MDDFACT ModularDistinctDegreeFactorizer

— ModularDistinctDegreeFactorizer.input —

)set break resume
)sys rm -f ModularDistinctDegreeFactorizer.output
)spool ModularDistinctDegreeFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModularDistinctDegreeFactorizer
--E 1

)spool
)lisp (bye)

________

— ModularDistinctDegreeFactorizer.help —

====================================================================
ModularDistinctDegreeFactorizer examples
====================================================================
This package supports factorization and gcds of univariate polynomials
over the integers modulo different primes. The inputs are given as
polynomials over the integers with the prime passed explicitly as an
extra argument.

See Also:
- )show ModularDistinctDegreeFactorizer

---

ModularDistinctDegreeFactorizer (MDDFACT)

Exports:
- factor
gcd
- linears
ddFact
- exptMod
- separateFactors

--- package MDDFACT ModularDistinctDegreeFactorizer ---

)abbrev package MDDFACT ModularDistinctDegreeFactorizer
++ Author: Barry Trager
++ Date Last Updated: 20.9.95 (JHD)
++ Description:
++ This package supports factorization and gcds
++ of univariate polynomials over the integers modulo different
++ primes. The inputs are given as polynomials over the integers
++ with the prime passed explicitly as an extra argument.

ModularDistinctDegreeFactorizer(U):C == T where
  U : UnivariatePolynomialCategory(Integer)
  I ==> Integer
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  V ==> Vector
L ==> List
DDRecord ==> Record(factor:EMR,degree:I)
UDDRecord ==> Record(factor:U,degree:I)
DDList ==> L DDRecord
UDDList ==> L UDDRecord

C == with
gcd:(U,U,I) -> U
  ++ gcd(f1,f2,p) computes the gcd of the univariate polynomials
  ++ f1 and f2 modulo the integer prime p.
lines: (U,I) -> U
  ++ linesars(f,p) returns the product of all the linear factors
  ++ of f modulo p. Potentially incorrect result if f is not
  ++ square-free modulo p.
factor:(U,I) -> L U
  ++ factor(f1,p) returns the list of factors of the univariate
  ++ polynomial f1 modulo the integer prime p.
  ++ Error: if f1 is not square-free modulo p.
ddFact:(U,I) -> UDDList
  ++ ddFact(f,p) computes a distinct degree factorization of the
  ++ polynomial f modulo the prime p, i.e. such that each factor
  ++ is a product of irreducibles of the same degrees. The input
  ++ polynomial f is assumed to be square-free modulo p.
separateFactors:(UDDList,I) -> L U
  ++ separateFactors(ddl,p) refines the distinct degree factorization
  ++ produced by ddFact to give a complete list of factors.
exptMod:(U,I,U,I) -> U
  ++ exptMod(f,n,g,p) raises the univariate polynomial f to the nth
  ++ power modulo the polynomial g and the prime p.

T == add
  zero? p => u
  map((i1:I):I +-> positiveRemainder(i1,p),u)
merge(p:I,q:I):Union(I,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"
modInverse(c:I,p:I):I ==
  (extendedEuclidean(c,p,1)::Record(coef1:I,coef2:I)).coef1
  invlcv:=modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlcv*v,p))
  reduction(r.remainder,p) ^=0 => "failed"
  reduction(invlcv*r.quotient,p)
EMR := EuclideanModularRing(Integer,U,Integer,
  reduction,merge,exactquo)
probSplit2:(EMR,EMR,I) -> Union(List EMR,"failed")
trace:(EMR,I,EMR) -> EMR
ddfactor:EMR -> L EMR
ddfact:EMR -> DDList
sepfact1:DDRecord -> L EMR
sepfact:DDList -> L EMR
probSplit:(EMR,EMR,I) -> Union(L EMR,"failed")
makeMonic:EMR -> EMR
exptmod:(EMR,I,EMR) -> EMR

lc(u:EMR):I == leadingCoefficient(u::U)
degree(u:EMR):I == degree(u::U)
makeMonic(u) == modInverse(lc(u),modulus(u)) * u

i:I
exptmod(u1,i,u2) ==
i < 0 => error("negative exponentiation not allowed for exptMod")
ans:= 1$EMR
while i > 0 repeat
  if odd?(i) then ans:= (ans * u1) rem u2
  i:= i quo 2
  u1:= (u1 * u1) rem u2
ans

exptMod(a,i,b,q) ==
ans:= exptmod(reduce(a,q),i,reduce(b,q))
ans::U

ddfactor(u) ==
if (c:= lc(u)) ^= 1$I then u:= makeMonic(u)
ans:= sepfact(ddfact(u))
cons(c::EMR,[makeMonic(f) for f in ans | degree(f) > 0])

gcd(u,v,q) == gcd(reduce(u,q),reduce(v,q))::U

factor(u,q) ==
v:= reduce(u,q)
dv:= reduce(differentiate(u),q)
degree gcd(v,dv) > 0 =>
  error("Modular factor: polynomial must be squarefree")
ans:= ddfactor v
[f::U for f in ans]

ddfact(u) ==
p:=modulus u
w:= reduce(monomial(1,1)$U,p)
m:= w
d:I:= 1
if (c:= lc(u)) ^= 1$I then u:= makeMonic u
ans:=DDList:=[]
repeat
  w:= exptmod(w,p,u)
  g:= gcd(w - m,u)
  if degree g > 0 then
    g:= makeMonic(g)
    ans:= [[g,d],:ans]
    u:= (u quo g)
    degree(u) = 0 => return [[c::EMR,0$I],:ans]
    d:= d+1
    d > (degree(u):I quo 2) =>
      return [[c::EMR,0$I],[u,degree(u)],:ans]
  
ddFact(u,q) ==
  ans:= ddfact(reduce(u,q))
  [[(dd.factor)::U,dd.degree]$UDDRecord for dd in ans]$UDDList

linears(u,q) ==
  uu:=reduce(u,q)
  m:= reduce(monomial(1,1)$U,q)
  gcd(exptmod(m,q,uu)-m,uu)::U

sepfact(factList) ==
  "append"/ [sepfact1(f) for f in factList]

separateFactors(uddList,q) ==
  ans:= sepfact [[reduce(udd.factor,q),udd.degree]$DDRecord for
    udd in uddList]$DDList
  [f::U for f in ans]

decode(s:Integer, p:Integer, x:U):U ==
  s<p => s::U
  qr := divide(s,p)
  qr.remainder :: U + x*decode(qr.quotient, p, x)

sepfact1(f) ==
  u:= f.factor
  p:=modulus u
  (d := f.degree) = 0 => [u]
  if (c:= lc(u)) ^= 1$I then u:= makeMonic(u)
  d = (du := degree(u)) => [u]
  ans:L EMR:=[ ]
  x:U:= monomial(1,1)
  -- for small primes find linear factors by exhaustion
  d=1 and p < 1000 =>
    for i in 0.. while du > 0 repeat
      if u(i::U) = 0 then
        ans := cons(reduce(x-(i::U),p),ans)
        du := du-1
  ans
y := x
s := 0
ss := 1
stack := [u]

until null stack repeat
  t := reduce(((s::U)+x), p)
  if not ((flist := probSplit(first stack, t, d)) case "failed") then
    stack := rest stack
    for fact in flist repeat
      f1 := makeMonic(fact)
      (df1 := degree(f1)) = 0 => nil
      df1 > d => stack := [f1, stack]
      ans := [f1, ans]
    end if
  end if
  p = 2 =>
    ss := ss + 1
  x := y * decode(ss, p, y)
  s := s + 1
  s := 0
  ss := ss + 1
  x := y * decode(ss, p, y)
  not one? leadingCoefficient(x) =>
    not (leadingCoefficient(x) = 1) =>
      ss := p ** degree x
      x := y ** (degree(x) + 1)
      ans := [c * first(ans), rest(ans)]
  end if
end for

probSplit(u, t, d) ==
  if (p := modulus(u)) = 2 then
    probSplit2(u, t, d)
  fi := gcd(u, t)
  r := ((p ** (d:NNI) - 1) quo 2):NNI
  n := exptmod(t, r, u)
  f2 := gcd(u, n + 1)
  (g := f1 * f2) = 1 => "failed"
  g = u => "failed"
  [f1, f2, (u quo g)]

probSplit2(u, t, d) ==
  f := gcd(u, trace(t, d, u))
  f = 1 => "failed"
  degree u = degree f => "failed"
  [1, f, u quo f]

trace(t, d, u) ==
  p := modulus(t)
  d := d - 1
  tt := t
  while d > 0 repeat
    tt := (tt + (t := exptmod(t, p, u))) rem u
    d := d - 1
package MHROWRED ModularHermitianRowReduction

— ModularHermitianRowReduction.input —

)set break resume
)sys rm -f ModularHermitianRowReduction.output
)spool ModularHermitianRowReduction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModularHermitianRowReduction
--E 1

)spool
)lisp (bye)

— ModularHermitianRowReduction.help —

=============================================
ModularHermitianRowReduction examples
=============================================

Modular hermitian row reduction.

See Also:
  o )show ModularHermitianRowReduction
ModularHermitianRowReduction (MHROWRED)

Exports:
rowEch  normalizedDivide  rowEchLocal  rowEchelon  rowEchelonLocal

— package MHROWRED ModularHermitianRowReduction —

)abbrev package MHROWRED ModularHermitianRowReduction
++ Author: Manuel Bronstein
++ Date Created: 22 February 1989
++ Date Last Updated: 24 November 1993
++ Description:
++ Modular hermitian row reduction.
-- should be moved into matrix whenever possible

ModularHermitianRowReduction(R):Exports == Implementation where
R: EuclideanDomain

Z ==> Integer
V ==> Vector R
M ==> Matrix R
REC ==> Record(val:R, cl:Z, rw:Z)

Exports ==> with
rowEch : M -> M
++ rowEch(m) computes a modular row-echelon form of m, finding
++ an appropriate modulus.
rowEchelon : (M, R) -> M
++ rowEchelon(m, d) computes a modular row-echelon form mod d of
++ [d ]
++ [ d ]
++ [ . ]
++ \[ \begin{bmatrix} d \\ M \end{bmatrix} \]
++ where \texttt{\spad{M = m mod d}}.

\texttt{rowEchLocal} : (M, R) \to M
++ \texttt{rowEchLocal(m, p)} computes a modular row-echelon form of \texttt{m}, finding
++ an appropriate modulus over a local ring where \texttt{p} is the only prime.

\texttt{rowEchelonLocal} : (M, R, R) \to M
++ \texttt{rowEchelonLocal(m, d, p)} computes the row-echelon form of \texttt{m}
++ concatenated with \texttt{d} times the identity matrix
++ over a local ring where \texttt{p} is the only prime.

\texttt{normalizedDivide: (R, R) \to Record(quotient:R, remainder:R)}
++ \texttt{normalizedDivide(n, d)} returns a normalized quotient and
++ remainder such that consistently unique representatives
++ for the residue class are chosen, e.g. positive remainders

\texttt{Implementation \rightarrow add}

\texttt{order : (R, R) \to Z}

\texttt{vconc : (M, R) \to M}

\texttt{non0 : (V, Z) \to Union(REC, \texttt{"failed"})}

\texttt{nonzero?: V \to Boolean}

\texttt{mkMat : (M, List Z) \to M}

\texttt{diagSubMatrix: M \to Union(Record(val:R, mat:M), \texttt{"failed"})}

\texttt{determinantOfMinor: M \rightarrow R}

\texttt{enumerateBinomial: (List Z, Z, Z) \rightarrow List Z}

\texttt{nonzero? v == any?(s +-> s ^= 0, v)}

\texttt{non0(v, rown) == ans:REC}

\texttt{allZero:Boolean := true}

\texttt{for i in minIndex v .. maxIndex v repeat}

\texttt{if qelt(v, i) ^= 0 then}

\texttt{if allZero then}

\texttt{allZero := false}

\texttt{ans := [qelt(v, i), i, rown]}

\texttt{else return \texttt{"failed"}}

\texttt{allZero => \texttt{"failed"}}

\texttt{ans}

\texttt{-- returns \[a, i, rown]\} if \(v = [0,...,0,a,0,...,0]\)

\texttt{-- where \(a \neq 0\) and \(i\) is the index of \(a\), \texttt{"failed"} otherwise.}

\texttt{non0(v, rown) == ans:REC}

\texttt{allZero:Boolean := true}

\texttt{for i in minIndex v .. maxIndex v repeat}

\texttt{if qelt(v, i) ^= 0 then}

\texttt{if allZero then}

\texttt{allZero := false}

\texttt{ans := [qelt(v, i), i, rown]}

\texttt{else return \texttt{"failed"}}

\texttt{allZero => \texttt{"failed"}}

\texttt{ans}

\texttt{-- returns a matrix made from the non-zero rows of \(x\) whose row number
-- is not in \(l\)}

\texttt{mkMat(x, l) == empty?(ll := \{parts row(x, i)

\texttt{for i in minRowIndex x .. maxRowIndex x | (not member?(i, l)) and nonzero? row(x, i)\}Thread(List R)) \Rightarrow}

\texttt{zero(1, ncols x)
matrix ll

-- returns \([m, d]\) where \(m = x\) with the zero rows and the rows of
-- the diagonal of \(d\) removed, if \(x\) has a diagonal submatrix of \(d\)'s,
-- "failed" otherwise.
diagSubMatrix x ==
  l := [u::REC for i in minRowIndex x .. maxRowIndex x |
    (u := nom0(row(x, i), i)) case REC]
  for a in removeDuplicates([r.val for r in l]$List(R)) repeat
    {[r.cl for r in l | r.val = a]$List(Z)}$Set(Z) =
    {[z for z in minColIndex x .. maxColIndex x]$List(Z)}$Set(Z)
    => return [a, mkMat(x, [r.rw for r in l | a = r.val])]
  "failed"

-- returns a non-zero determinant of a minor of \(x\) of rank equal to
-- the number of columns of \(x\), if there is one, 0 otherwise
determinantOfMinor x ==
  -- do not compute a modulus for square matrices, since this is as expensive
  -- as the Hermite reduction itself
  (nr := nrows x) <= (nc := ncols x) => 0
  lc := [i for i in minColIndex x .. maxColIndex x]$List(Integer)
  lr := [i for i in minRowIndex x .. maxRowIndex x]$List(Integer)
  for i in 1..(n := binomial(nr, nc)) repeat
    d := determinant x(enumerateBinomial(lr, nc, i), lc)
    ^= 0 =>
    j := i + 1 + (random()$Z rem (n - i))
    return gcd(d, determinant x(enumerateBinomial(lr, nc, j), lc))
  0

-- returns the \(i\)-th selection of \(m\) elements of \(l = (a_1, \ldots, a_n)\),
-- \(\begin{array}{c}
  \left/\begin{array}{l}
  \text{\(1 \leq i \leq |l|\)}
  \hline
  \text{\(\begin{array}{l}
  a_1
  \end{array}\)}
  \end{array}\right\end{array}\)
--
enumerateBinomial(l, m, i) ==
  m1 := minIndex l - 1
  zero?(m := m - 1) => [l(m1 + i)]
  for j in 1..(n := #l) repeat
    i <= (b := binomial(n - j, m)) =>
    return concat(l(m1 + j), enumerateBinomial(rest(l, j), m, i))
    i := i - b
  error "Should not happen"

rowEch x ==
  (u := diagSubMatrix x) case "failed" =>
  zero?(d := determinantOfMinor x) => rowEchelon x
  rowEchelon(x, d)
  rowEchelon(u.mat, u.val)

vconc(y, m) ==
  vertConcat(diagonalMatrix new(ncols y, m)$V, map(s -> s rem m, y))
order(m, p) ==
  zero? m => -1
  for i in 0.. repeat
    (mm := m exquo p) case "failed" => return i
    m := mm::R
  if R has IntegerNumberSystem then
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      qr := divide(n, d)
      qr.remainder >= 0 => qr
      d > 0 =>
        qr.remainder := qr.remainder + d
        qr.quotient := qr.quotient - 1
        qr
      qr.remainder := qr.remainder - d
      qr.quotient := qr.quotient + 1
      qr
  else
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      divide(n, d)
  end if

rowEchLocal(x, p) ==
  (u := diagSubMatrix x) case "failed" =>
  zero?(d := determinantOfMinor x) => rowEchelon x
  rowEchelonLocal(x, d, p)
  rowEchelonLocal(u.mat, u.val, p)

rowEchelonLocal(y, m, p) ==
  m := p**(order(m,p)::NonNegativeInteger)
  x := vconc(y, m)
  nrows := maxRowIndex x
  ncols := maxColIndex x
  minr := i := minRowIndex x
  for j in minColIndex x .. ncols repeat
    if i > nrows then leave x
    rown := minr - 1
    pivord : Integer
    npivord : Integer
    for k in i .. nrows repeat
      qelt(x,k,j) = 0 => "next k"
      npivord := order(qelt(x,k,j),p)
      (rown = minr - 1) or (npivord < pivord) =>
        rown := k
        pivord := npivord
    end for
    rown = minr - 1 => "enuf"
    x := swapRows_!(x, i, rown)
    (a, b, d) := extendedEuclidean(qelt(x,i,j), m)
    qsetelt_!(x,i,j,d)
    pivot := d
  end for
  for k in j+1 .. ncols repeat
CHAPTER 14. CHAPTER M

qsetelt_!(x, i, k, a * qelt(x, i, k) rem m)
for k in i+1 .. nrows repeat
  zero? qelt(x, k, j) => "next k"
  q := (qelt(x, k, j) exquo pivot) :: R
for k1 in j+1 .. ncols repeat
  v2 := (qelt(x, k, k1) - q * qelt(x, i, k1)) rem m
  qsetelt_!(x, k, k1, v2)
qsetelt_!(x, k, j, 0)
for k in minr .. i-1 repeat
  zero? qelt(x, k, j) => "enuf"
  qr := normalizedDivide(qelt(x, k, j), pivot)
qsetelt_!(x, k, j, qr.quotient * qelt(x, i, k1) rem m)
i := i+1
x

if R has Field then
  rowEchelon(y, m) == rowEchelon vconc(y, m)
else

  rowEchelon(y, m) ==
  x := vconc(y, m)
nrows := maxRowIndex x
ncols := maxColIndex x
minr := i := minRowIndex x
for j in minColIndex x .. ncols repeat
  if i > nrows then leave
  rown := minr - 1
  for k in i .. nrows repeat
    if (qelt(x, k, j) ^= 0) and ((rown = minr - 1) or sizeLess?(qelt(x, k, j), qelt(x, rown, j))) then rown := k
  rown = minr - 1 => "next j"
  x := swapRows_!(x, i, rown)
for k in i+1 .. nrows repeat
  zero? qelt(x, k, j) => "next k"
  (a, b, d) := extendedEuclidean(qelt(x, i, j), qelt(x, k, j))
  (b1, a1) :=
    ((qelt(x, i, j) exquo d)::R, (qelt(x, k, j) exquo d)::R)
    -- a*b1+a1*b = 1
  for k1 in j+1 .. ncols repeat
    v1 := (a * qelt(x, i, k1) + b * qelt(x, k, k1)) rem m
    v2 := (b1 * qelt(x, k, k1) - a1 * qelt(x, i, k1)) rem m
    qsetelt_!(x, i, k1, v1)
    qsetelt_!(x, k, k1, v2)
    qsetelt_!(x, i, j, d)
    qsetelt_!(x, k, j, 0)
  un := unitNormal qelt(x, i, j)
qsetelt_!(x,i,j,un.canonical)
if un.associate ^= 1 then for jj in (j+1)...ncols repeat
  qsetelt_!(x,i,jj,un.associate * qelt(x,i,jj))

xij := qelt(x,i,j)
for k in minr .. i-1 repeat
  zero? qelt(x,k,j) => "next k"
  qr     := normalizedDivide(qelt(x,k,j), xij)
  qsetelt_!(x,k,j, qr.remainder)
for k1 in j+1 .. ncols x repeat
  qsetelt_!(x,k,k1,
    (qelt(x,k,k1) - qr.quotient * qelt(x,i,k1)) rem m)
i := i+1
x

— MHROWRED.dotabb —
"MHROWRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MHROWRED"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"MHROWRED" -> "IVECTOR"

——
package MRF2 MonoidRingFunctions2

--- MonoidRingFunctions2.input ---

)set break resume
)sys rm -f MonoidRingFunctions2.output
)spool MonoidRingFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MonoidRingFunctions2
--E 1

)spool
)lisp (bye)

——
MonoidRingFunctions2 (MRF2)

Exports:

map

--- package MRF2 MonoidRingFunctions2 ---

)abbrev package MRF2 MonoidRingFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 14 May 1991
++ Date Last Updated: 14 May 1991
++ Description:
++ MonoidRingFunctions2 implements functions between
++ two monoid rings defined with the same monoid over different rings.

MonoidRingFunctions2(R,S,M) : Exports == Implementation where
R : Ring
S : Ring
M : Monoid
Exports ==> with
  map: (R -> S, MonoidRing(R,M)) -> MonoidRing(S,M)
  ++ map(f,u) maps f onto the coefficients f the element
  ++ u of the monoid ring to create an element of a monoid
  ++ ring with the same monoid b.
Implementation ==> add
map(fn, u) ==
  res : MonoidRing(S,M) := 0
  for te in terms u repeat
    res := res + monomial(fn(te.coef), te.monom)
  res

|---

MRF2.dotabb

"MRF2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MRF2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"MRF2" -> "LMODULE"
"MRF2" -> "SGROUP"

|---

package MONOTOOL MonomialExtensionTools

--- MonomialExtensionTools.input ---

)set break resume
)sys rm -f MonomialExtensionTools.output
)spool MonomialExtensionTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MonomialExtensionTools
--E 1

)spool
)lisp (bye)

|---
MonomialExtensionTools examples

Tools for handling monomial extensions.

See Also:
- )show MonomialExtensionTools

Exports:
- decompose
- normalDenom
- split
- splitSquarefree

---

MonomialExtensionTools (MONOTOOL)

Exports:
- decompose
- normalDenom
- split
- splitSquarefree

---

)abbrev package MONOTOOL MonomialExtensionTools
++ Author: Manuel Bronstein
++ Date Created: 18 August 1992
++ Date Last Updated: 3 June 1993
++ Description:
++ Tools for handling monomial extensions.

MonomialExtensionTools(F, UP): Exports == Implementation where
  F : Field
  UP: UnivariatePolynomialCategory F
  RF ==> Fraction UP
FR ==> Factored UP

Exports ==> with
split : (UP, UP -> UP) -> Record(normal:UP, special:UP)
  ++ split(p, D) returns \spad{\{n, s\}} such that \spad{p = n s},
  ++ all the squarefree factors of n are normal w.r.t. D,
  ++ and s is special w.r.t. D.
  ++ D is the derivation to use.

splitSquarefree: (UP, UP -> UP) -> Record(normal:FR, special:FR)
  ++ splitSquarefree(p, D) returns
  ++ \spad{\{n_1 n_2\^2 \ldots n_m\^m, s_1 s_2\^2 \ldots s_q\^q\}} such that
  ++ \spad{p = n_1 n_2\^2 \ldots n_m\^m s_1 s_2\^2 \ldots s_q\^q}, each
  ++ \spad{n_i} is normal w.r.t. D and each \spad{s_i} is special
  ++ w.r.t. D.
  ++ D is the derivation to use.

normalDenom: (RF, UP -> UP) -> UP
  ++ normalDenom(f, D) returns the product of all the normal factors
  ++ of \spad{denom(f)}.
  ++ D is the derivation to use.

decompose : (RF, UP -> UP) -> Record(poly:UP, normal:RF, special:RF)
  ++ decompose(f, D) returns \spad{\{p, n, s\}} such that \spad{f = p+n+s},
  ++ all the squarefree factors of \spad{denom(n)} are normal w.r.t. D,
  ++ \spad{denom(s)} is special w.r.t. D,
  ++ and n and s are proper fractions (no pole at infinity).
  ++ D is the derivation to use.

Implementation ==> add

normalDenom(f, derivation) == split(denom f, derivation).normal

split(p, derivation) ==
pbar := (gcd(p, derivation p) exquo gcd(p, differentiate p))::UP
zero? degree pbar => [p, 1]
rec := split((p exquo pbar)::UP, derivation)
[rec.normal, pbar * rec.special]

splitSquarefree(p, derivation) ==
s:Factored(UP) := 1
n := s
q := squareFree p
for rec in factors q repeat
  r := rec.factor
g := gcd(r, derivation r)
  if not ground? g then s := s * sqfrFactor(g, rec.exponent)
  h := (r exquo g)::UP
  if not ground? h then n := n * sqfrFactor(h, rec.exponent)
[n, unit(q) * s]

decompose(f, derivation) ==
qr := divide(numer f, denom f)
-- rec.normal * rec.special = denom f
rec := split(denom f, derivation)
-- eeu.coef1 * rec.normal + eeu.coef2 * rec.special = qr.remainder
-- and degree(eeu.coef1) < degree(rec.special)
-- and degree(eeu.coef2) < degree(rec.normal)
-- qr.remainder/denom(f) = eeu.coef1 / rec.special + eeu.coef2 / rec.normal
  eeu := extendedEuclidean(rec.normal, rec.special, qr.remainder)::Record(coef1:UP, coef2:UP)
  [qr.quotient, eeu.coef2 / rec.normal, eeu.coef1 / rec.special]

---

package MSYSCMD MoreSystemCommands

--- MoreSystemCommands.input ---

)set break resume
)sys rm -f MoreSystemCommands.output
)spool MoreSystemCommands.output
)set message test on
)set message auto off
)clear all

--) 1 of 1
)show MoreSystemCommands
--E 1

)spool
)lisp (bye)

---

--- MoreSystemCommands.help ---

====================================================================
MoreSystemCommands examples
MoreSystemCommands implements an interface with the system command facility. These are the commands that are issued from source files or the system interpreter and they start with a close parenthesis, e.g., the "what" commands.

See Also:
o )show MoreSystemCommands

MoreSystemCommands (MSYSCMD)

Exports:
  systemCommand

— package MSYSCMD MoreSystemCommands —

)abbrev package MSYSCMD MoreSystemCommands
++ Description:
++ \spadtype{MoreSystemCommands} implements an interface with the
++ system command facility. These are the commands that are issued
++ from source files or the system interpreter and they start with
++ a close parenthesis, for example, the "what" commands.

MoreSystemCommands: public == private where

public == with

    systemCommand: String -> Void
    ++ systemCommand(cmd) takes the string \spadvar{cmd} and passes
    ++ it to the runtime environment for execution as a system
++ command. Although various things may be printed, no usable
++ value is returned.

private == add

systemCommand cmd == doSystemCommand(cmd)$Lisp

---

— MSYSCMD.dotabb —

"MSYSCMD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MSYSCMD"]
"Package" [color="#FF4488"]
"MSYSCMD" -> "Package"

---

package MPCPF MPolyCatPolyFactorizer

— MPolyCatPolyFactorizer.input —

)set break resume
/sys rm -f MPolyCatPolyFactorizer.output
)spool MPolyCatPolyFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatPolyFactorizer
--E 1

)spool
)lisp (bye)

---

— MPolyCatPolyFactorizer.help —

====================================================================
MPolyCatPolyFactorizer examples
====================================================================
This package exports a factor operation for multivariate polynomials with coefficients which are polynomials over some ring \( R \) over which we can factor. It is used internally by packages such as the solve package which need to work with polynomials in a specific set of variables with coefficients which are polynomials in all the other variables.

See Also:
- \( \)show MPolyCatPolyFactorizer

---

MPolyCatPolyFactorizer (MPCPF)

Exports:
- factor

---

\( \)abbrev package MPCPF MPolyCatPolyFactorizer
++ Author: P. Gianni
++ Date Last Updated: March 1995
++ Description:
++ This package exports a factor operation for multivariate polynomials with coefficients which are polynomials over some ring \( R \) over which we can factor. It is used internally by packages such as the solve package which need to work with polynomials in a specific set of variables with coefficients which are polynomials in all the other variables.

\( \)MPolyCatPolyFactorizer(E,OV,R,PPR) : C == T

where
- \( R \) : EuclideanDomain
CHAPTER 14. CHAPTER M

E : OrderedAbelianMonoidSup
   -- following type is required by PushVariables
OV : OrderedSet with
   convert : % -> Symbol
      ++ convert(x) converts x to a symbol
variable: Symbol -> Union(%,”failed”)
      ++ variable(s) makes an element from symbol s or fails.
PR ==> Polynomial R
PPR : PolynomialCategory(PR,E,OV)
NNI ==> NonNegativeInteger
ISY ==> IndexedExponents Symbol
SE ==> Symbol
UP ==> SparseUnivariatePolynomial PR
UPPR ==> SparseUnivariatePolynomial PPR

C == with
   factor : PPR -> Factored PPR
      ++ factor(p) factors a polynomial with polynomial
      ++ coefficients.

--- Local Functions ----
T == add

import PushVariables(R,E,OV,PPR)

---- factorization of p ----
factor(p:PPR) : Factored PPR ==
ground? p => nilFactor(p,1)
c := content p
p := (p exquo c)::PPR
vars:List OV := variables p
g:PR:=retract pushdown(p, vars)
flist := factor(g)$GeneralizedMultivariateFactorize(Symbol,ISY,R,R,PR)
ffact : List(Record(irr:PPR,pow:Integer))
   ffact:=[pushup(u.factor::PPR,vars),u.exponent] for u in factors flist
   fcont:=(unit flist)::PPR
   nilFactor(c*fcont,1)*(_*/[primeFactor(ff.irr,ff.pow) for ff in ffact])

———

— MPCPF.dotabb ——

"MPCPF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPCPF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPCPF" -> "PFECAT"
package MPRFF MPolyCatRationalFunctionFactorizer

--- MPolyCatRationalFunctionFactorizer.input ---

)set break resume
/sys rm -f MPolyCatRationalFunctionFactorizer.output
/spool MPolyCatRationalFunctionFactorizer.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show MPolyCatRationalFunctionFactorizer
--E 1

)spool
)lisp (bye)

--- MPolyCatRationalFunctionFactorizer.help ---

====================================================================
MPolyCatRationalFunctionFactorizer examples
====================================================================

This package exports a factor operation for multivariate polynomials
with coefficients which are rational functions over some ring R over
which we can factor. It is used internally by packages such as primary
decomposition which need to work with polynomials with rational
function coefficients, i.e. themselves fractions of polynomials.

See Also:
o )show MPolyCatRationalFunctionFactorizer
CHAPTER 14. CHAPTER M

MPolyCatRationalFunctionFactorizer (MPRFF)

Exports:

- factor
- pushdown
- pushup
- pushdterm
- pushucoef
- pushuconst
- totalfract

-- package MPRFF MPolyCatRationalFunctionFactorizer --

)abbrev package MPRFF MPolyCatRationalFunctionFactorizer
++ Author: P. Gianni
++ Description:
++ This package exports a factor operation for multivariate polynomials
++ with coefficients which are rational functions over
++ some ring R over which we can factor. It is used internally by packages
++ such as primary decomposition which need to work with polynomials
++ with rational function coefficients, i.e. themselves fractions of
++ polynomials.

MPolyCatRationalFunctionFactorizer(E,OV,R,PRF) : C == T
where

R : IntegralDomain
F ==> Fraction Polynomial R
RN ==> Fraction Integer
E : OrderedAbelianMonoidSup
OV : OrderedSet with
  convert : % -> Symbol
  ++ convert(x) converts x to a symbol
PRF : PolynomialCategory(F,E,OV)
NNI ==> NonNegativeInteger
P ==> Polynomial R
ISE ==> IndexedExponents SE
SE ==> Symbol
UP ==> SparseUnivariatePolynomial P
UF ==> SparseUnivariatePolynomial F
UPRF ==> SparseUnivariatePolynomial PRF
QuoForm ==> Record(sup:P,inf:P)
C == with
  totalfract : PRF -> QuoForm
    ++ totalfract(prf) takes a polynomial whose coefficients are
    ++ themselves fractions of polynomials and returns a record
    ++ containing the numerator and denominator resulting from
    ++ putting prf over a common denominator.
  pushdown : (PRF,OV) -> PRF
    ++ pushdown(prf, var) pushes all top level occurences of the
    ++ variable var into the coefficient domain for the polynomial prf.
  pushdterm : (UPRF,OV) -> PRF
    ++ pushdterm(monom, var) pushes all top level occurences of the
    ++ variable var into the coefficient domain for the monomial monom.
  pushup : (PRF,OV) -> PRF
    ++ pushup(prf, var) raises all occurrences of the
    ++ variable var in the coefficients of the polynomial prf
    ++ back to the polynomial level.
  pushucoef : (UP,OV) -> PRF
    ++ pushucoef(upoly, var) converts the anonymous univariate
    ++ polynomial upoly to a polynomial in var over rational functions.
  pushuconst : (F,OV) -> PRF
    ++ pushuconst(r, var) takes a rational function and raises
    ++ all occurrences of the variable var to the polynomial level.
  factor : PRF -> Factored PRF
    ++ factor(prf) factors a polynomial with rational function
    ++ coefficients.

--- Local Functions ---
T == add

    ---- factorization of p ----
factor(p:PRF) : Factored PRF ==
traelist:List OV := variables p
  tp:=totalfract(p)
nump:P:= tp.sup
denp:F:=inv(tp.inf ::F)
  ffact : List(Record(irr:PRF,pow:Integer))
  flist:Factored P
  if R is Fraction Integer then
    flist:=
      ((factor nump)$MRationalFactorize(ISE,SE,Integer,P))
        pretend (Factored P)
  else
    if R has FiniteFieldCategory then
      flist:= ((factor nump)$MultFiniteFactorize(SE,ISE,R,P))
        pretend (Factored P)
    else
      if R has Field then error "not done yet"
      else
        flist:
if $R$ has CharacteristicZero then
  \[\text{flist} := ((\text{factor nump}) \text{MultivariateFactorize}(\text{SE}, \text{ISE}, R, P))\]
  \text{pretend (Factored P)}
else error "can't happen"

\[\text{ffact} := \text{[}\text{[}\text{factor}\text{.}\text{factor}::F::PRF, \text{u}\text{.}\text{exponent} \text{for}\text{ u}\text{ in}\text{ factors}\text{ flist}\text{]}\text{]}\text{]}\]
\text{fcont} := (\text{unit}\text{ flist})::F::PRF

\text{for}\text{ x}\text{ in true\text{list}\ repeat}
\text{fcont} := \text{pushup}(\text{fcont}, x)

\text{ffact} := \text{[}\text{[}\text{pushup}(\text{ff}\text{.}\text{irr}, x), \text{ff}\text{.}\text{pow} \text{for}\text{ ff}\text{ in}\text{ ffact}\text{]}\text{]}\text{]}\]
\text{(denp}\text{fcont})\text{*}(\text{_*}/\text{[primeFactor}(\text{ff}\text{.}\text{irr}, \text{ff}\text{.}\text{pow}) \text{for}\text{ ff}\text{ in}\text{ ffact}\text{]}\text{])}

\text{---- the following functions are used to "push" x in the coefficient ring ----}

\text{----- push x in the coefficient domain for a polynomial ----}
\text{pushdown}(g: \text{PRF}, x: \text{OV}) : \text{PRF} ==
\text{ground? g} \Rightarrow \text{g}
\text{rf:PRF:=0$PRF}
\text{ug:=univariate(g,x)}
\text{while ug} \neq 0 \text{ repeat}
\text{rf:=rf+pushdterm(ug,x)}
\text{ug := reductum ug}
\text{rf}

\text{----- push x in the coefficient domain for a term ----}
\text{pushdterm}(t: \text{UPRF}, x: \text{OV}): \text{PRF} ==
\text{n:=degree(t)}
\text{cf:=monomial(1,convert x,n)$P :: F}
\text{cf * leadingCoefficient t}

\text{----- push back the variable ----}
\text{pushup}(f: \text{PRF}, x: \text{OV}) : \text{PRF} ==
\text{ground? f} \Rightarrow \text{pushuconst(retract f, x)}
\text{v:=mainVariable(f)::OV}
\text{g:=univariate(f,v)}
\text{multivariate(map((y:PRF):PRF +-> pushup(y,x),g),v)}

\text{----- push x back from the coefficient domain ----}
\text{pushuconst}(r:F,x:OV):PRF ==
\text{xs:SE:=convert x}
\text{degree(denom r,xs)>0} \Rightarrow \text{error "bad polynomial form"}
\text{inv((denom r)::F)*pushucoef(univariate(numer r,xs),x)}

\text{pushucoef(c:UP,x:OV):PRF ==}
\text{c = 0} \Rightarrow \text{0}
\text{monomial(}((\text{leadingCoefficient c}):F::PRF,x,\text{degree c}) +
\text{pushucoef(reductum c,x))}
### totalfrac

```lisp
totalfrac(p:PRF) : QuoForm ==
p=0 => [0$P,1$P]$QuoForm
for x in variables p repeat p:=pushdown(p,x)
g:F:=retract p
[numer g,denom g]$QuoForm
```

---

### package MPC2 MPolyCatFunctions2

```lisp
package MPC2 MPolyCatFunctions2

)spool MPolyCatFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatFunctions2
--E 1

)spool
)lisp (bye)
```

---

### MPolyCatFunctions2.help

```lisp
====================================================================
MPolyCatFunctions2 examples
====================================================================
```

---
Utilities for MPolyCat

See Also:
- )show MPolyCatFunctions2

MPolyCatFunctions2 (MPC2)

Exports:
- map
- reshape

-- package MPC2 MPolyCatFunctions2 --

)abbrev package MPC2 MPolyCatFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 28 March 1990 (PG)
++ Description:
++ Utilities for MPolyCat

MPolyCatFunctions2(VarSet,E1,E2,R,PR,PS) : public == private where

VarSet : OrderedSet
E1 : OrderedAbelianMonoidSup
E2 : OrderedAbelianMonoidSup
R : Ring
S : Ring
PR : PolynomialCategory(R,E1,VarSet)
PS : PolynomialCategory(S,E2,VarSet)
SUPR ==> SparseUnivariatePolynomial PR
SUPS ==> SparseUnivariatePolynomial PS
public == with
  map: (R -> S, PR) -> PS
  ++ map(f,p) \undocumented
reshape: (List S, PR) -> PS
  ++ reshape(l,p) \undocumented

private == add
  supMap: (R -> S, SUPR) -> SUPS

supMap(fn : R -> S, supr : SUPR): SUPS ==
  supr = 0 => monomial(fn(0$R) :: PS,0)$SUPS
  c : PS := map(fn,leadingCoefficient supr)$%
  monomial(c,degree supr)$SUPS + supMap(fn, reductum supr)

map(fn : R -> S, pr : PR): PS ==
  varu : Union(VarSet,"failed") := mainVariable pr
  varu case "failed" => -- have a constant
     fn(retract pr) :: PS
  var : VarSet := varu :: VarSet
  supr : SUPR := univariate(pr,var)$PR
  multivariate(supMap(fn,supr),var)$PS

---

MPC2.dotabb

"MPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC2" -> "PFECAT"

---

package MPC3 MPolyCatFunctions3

--- MPolyCatFunctions3.input ---

)set break resume
)sys rm -f MPolyCatFunctions3.output
)spool MPolyCatFunctions3.output
)set message test on
)set message auto off
)clear all
MPolyCatFunctions3 examples

This package has no description

See Also:
- )show MPolyCatFunctions3

Exports:
map

- package MPC3 MPolyCatFunctions3 -
++ This package has no description

MPolyCatFunctions3(Vars1, Vars2, E1, E2, R, PR1, PR2): C == T where
  E1 : OrderedAbelianMonoidSup
  E2 : OrderedAbelianMonoidSup
  Vars1: OrderedSet
  Vars2: OrderedSet
  R : Ring
  PR1 : PolynomialCategory(R, E1, Vars1)
  PR2 : PolynomialCategory(R, E2, Vars2)

C ==> with
  map: (Vars1 -> Vars2, PR1) -> PR2
  ++ map(f, x) undocumented

T ==> add

map(f: Vars1 -> Vars2, p: PR1): PR2 ==
  (x1 := mainVariable p) case "failed" =>
    c := retract p
    c : PR2
  up := univariate(p, x1:: Vars1)
  x2 := f(x1:: Vars1)
  ans: PR2 := 0
  while up ^= 0 repeat
    ans := ans + monomial(map(f, leadingCoefficient up), x2, degree up)
    up := reductum up
  ans

———

— MPC3.dotabb —

"MPC3" [color="#FF4488", href="bookvol10.4.pdf#nameddest=MPC3"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC3" -> "PFECAT"

———

package MRATFAC MRationalFactorize

— MRationalFactorize.input —

)set break resume
MRationalFactorize (MRATFAC)

Exports:
factor
package MRATFAC MRationalFactorize

++) abbrev package MRATFAC MRationalFactorize
++ Author: P. Gianni
++ Description:
++ MRationalFactorize contains the factor function for multivariate
++ polynomials over the quotient field of a ring R such that the package
++ MultivariateFactorize can factor multivariate polynomials over R.

MRationalFactorize(E,OV,R,P) : C == T
where
  E : OrderedAbelianMonoidSup
  OV : OrderedSet
  R : Join(EuclideanDomain, CharacteristicZero) -- with factor over R[x]
  FR ==> Fraction R
  P : PolynomialCategory(FR,E,OV)
  MPR ==> SparseMultivariatePolynomial(R,OV)
  SUP ==> SparseUnivariatePolynomial

C == with
  factor : P -> Factored P
  ++ factor(p) factors the multivariate polynomial p with coefficients
  ++ which are fractions of elements of R.

T == add
  IE ==> IndexedExponents OV
  PCLFRR ==> PolynomialCategoryLifting(E,OV,FR,P,MPR)
  PCLRFR ==> PolynomialCategoryLifting(IE,OV,R,MPR,P)
  MFACT ==> MultivariateFactorize(OV,IE,R,MPR)
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2

numer1(c:FR): MPR == (numer c) :: MPR
numer2(pol:P) : MPR == map(coerce,numer1,po$PCLFRR
coerce1(d:R) : P == (d::FR)::P
coerce2(pp:MPR) :P == map(coerce,coerce1,pp)$PCLFR

factor(p:P) : Factored P ==
  pden:R:=lcm([denom c for c in coefficients p])
  pol :P:= (pdens::FR)\*p
  ipol:MPR:= map(coerce,numer1,po$PCLFRR
  ffact:=(factor ipol)$MFACT
  (1/pden)\*map(coerce,coerce1,(unit ffact))$PCLFRF *
  _*/[primeFactor(map(coerce,coerce1,po.factor)$PCLFRF,
    u.exponent) for u in factors ffact]

---
package MFINFACT MultFiniteFactorize

--- MultFiniteFactorize.input ---

)set break resume
)sys rm -f MultFiniteFactorize.output
)spool MultFiniteFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultFiniteFactorize
--E 1

)spool
)lisp (bye)

--- MultFiniteFactorize.help ---

====================================================================
MultFiniteFactorize examples
====================================================================

Package for factorization of multivariate polynomials over finite fields.

See Also:
  o )show MultFiniteFactorize
MultFiniteFactorize (MFINFACT)

Exports:
factor

--- package MFINFACT MultFiniteFactorize ---

)abbrev package MFINFACT MultFiniteFactorize
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: 19 March 1992
++ Description:
++ Package for factorization of multivariate polynomials over finite fields.

MultFiniteFactorize(OV,E,F,PG) : C == T

where
F : FiniteFieldCategory
OV : OrderedSet
E : OrderedAbelianMonoidSup
PG : PolynomialCategory(F,E,OV)
SUP ==> SparseUnivariatePolynomial
R ==> SUP F
P ==> SparseMultivariatePolynomial(R,OV)
Z ==> Integer
FFPOLY ==> FiniteFieldPolynomialPackage(F)
MParFact ==> Record(irr:P,pow:Z)
MFinalFact ==> Record(contp:R,factors:List MParFact)
SUParFact ==> Record(irr:SUP P,pow:Z)
SUPFinalFact ==> Record(contp:R,factors:List SUParFact)

-- contp = content,
-- factors = List of irreducible factors with exponent

C == with

factor : PG -> Factored PG
++ factor(p) produces the complete factorization of the multivariate
++ polynomial p over a finite field.
factor : SUP PG -> Factored SUP PG
++ factor(p) produces the complete factorization of the multivariate
++ polynomial p over a finite field. p is represented as a univariate
++ polynomial with multivariate coefficients over a finite field.

T == add

import LeadingCoefDetermination(OV,IndexedExponents OV,R,P)
import MultivariateLifting( IndexedExponents OV,OV,R,P)
import FactoringUtilities( IndexedExponents OV,OV,R,P)
import FactoringUtilities(E,OV,F,PG)
import GenExEuclid(R,SUP R)

NNI ==> NonNegativeInteger
L ==> List
UPCF2 ==> UnivariatePolynomialCategoryFunctions2
LeadFact ==> Record(polfac:L P,correct:R,corrfact:L SUP R)
ContPrim ==> Record(cont:P,prim:P)
ParFact ==> Record( irr:_SUP R,pow:Z)
FinalFact ==> Record(contp:R,factors:L ParFact)
NewOrd ==> Record(npol:_SUP P,nvar:L OV,newdeg:L NNI)

---- Local Functions ----
ran : Z -> R
mFactor : (P,Z) -> MFinalFact
supFactor : (SUP P,Z) -> SUPFinalFact
mfconst : (SUP P,Z,L OV,L NNI) -> L SUP P
mpol : (SUP P,Z,L OV,L N NI) -> L SUP P
varChoose : (P,L OV,L NNI) -> NewOrd
simplify : (P,Z,L OV,L NNI) -> MFinalFact
intChoose : (SUP P,L OV,R,L P,L L R) -> Valuf
pretest : (P,NNI,L OV,L R) -> FinalFact
checkzero : (SUP P,SUP R) -> Boolean
pushdcoef : PG -> P
pushdown : (PG,OV) -> P
pushupconst : (R,OV) -> PG
pushup : (P,OV) -> PG
norm : L SUP R -> Integer
constantCase : (P,L MParFact) -> MFinalFact
pM : L SUP R -> R
intfact : (SUP P,L OV,L NNI,MFinalFact,L L R) -> L SUP P

basicVar:OV:=NIL$Lisp pretend OV -- variable for the basic step

convertPUP(lfg:MFinalFact): SUPFinalFact ==
[lfg.contp,[[lff.irr ::SUP P,lff.pow]$SUParFact
  for lff in lfg.factors]$SUPFinalFact
supFactor (um: SUP P, dx: Z) : SUPFinalFact ==
  degree (um) = 0 => convertPUP (mFactor (ground um, dx))
lvar: L OV := "setUnion" /[variables cf for cf in coefficients um]
lcont: SUP P
if: L SUP P

flead : SUPFinalFact := [0, empty()]$SUPFinalFact
factorlist: L SUParFact := empty()

mdeg := minimumDegree um  ---- is the Mindeg > 0? ----
if mdeg > 0 then
  f1: SUP P := monomial (1, mdeg)
  um := (um exquo f1) :: SUP P
  factorlist := cons ([monomial (1, 1), mdeg], factorlist)
  if degree um = 0 then return
  lfg := convertPUP mFactor (ground um, dx)
  [lfg.contp, append (factorlist, lfg.factors)]

om := map ((p1: P) PG -- > pushup (p1, basicVar), um)$UPCF2 (P, SUP P, PG, SUP PG)
sqfacs := squareFree (om)
lcont :=
  map ((p1: PG): P -- > pushdown (p1, basicVar), unit sqfacs) _
    $UPCF2 (PG, SUP PG, P, SUP P)

    ---- Factorize the content ----

if ground? lcont then
  flead := convertPUP constantCase (ground lcont, empty())
else
  flead := supFactor (lcont, dx)

factorlist := flead.factors

    ---- Make the polynomial square-free ----

sqqfact := [[map ((p: PG): P -- > pushdown (p, basicVar), ff.factor), ff.exponent]
  for ff in factors sqfacs]

    --- Factorize the primitive square-free terms ---

for fact in sqqfact repeat
  ffactor: SUP P := fact. irr
  ffexp := fact. pow
  ffcont := content ffactor
  coefs := coefficients ffactor
  ldeg := ["max" /[degree (fc, xx) for fc in coefs] for xx in lvar]
  if ground? (leadingCoefficient ffactor) then
    lf := mfconst (ffactor, dx, lvar, ldeg)
  else lf := mfpol (ffactor, dx, lvar, ldeg)
auxfl := [[lf[p, ffexp]]$SUParFact for lfp in lf]
factorlist := append (factorlist, auxfl)
lcfacs := 
  */[leadingCoefficient leadingCoefficient(f.irr)**((f.pow)::NNI)
    for f in factorlist]
[(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R,
  factorlist]$

factorlist[1,empty()]

mFactor(m:P,dx:Z) : MFinalFact ==
  ground?(m) => constantCase(m,empty())
  lvar:L OV:= variables m
  lcont:P
  lfac:L SUP P
  flead : MFinalFact:=[1,empty()]$MFinalFact
  factorlist:L MParFact :=empty()

  ---- is the Mindeg > 0? ----
  lmdeg :=minimumDegree(m,lvar)
  or/[n>0 for n in lmdeg] => simplify(m,dx,lvar,lmdeg)
  ---- Make the polynomial square-free ----
  om:=pushup(m,basicVar)
  sqfacs:=squareFree(om)
  lcont := pushdown(unit sqfacs,basicVar)

  ---- Factorize the content ----
  if ground? lcont then
    flead:=constantCase(lcont,empty())
  else
    flead:=mFactor(lcont,dx)
    factorlist:=flead.factors
    sqqfact:List Record(factor:P,exponent:Integer)
    sqqfact:=[pushdown(ff.factor,basicVar),ff.exponent]
      for ff in factors sqfacs]    
  ---- Factorize the primitive square-free terms ----
  for fact in sqqfact repeat
    ffactor:P:=fact.factor
    ffexp := fact.exponent
    ground? ffactor =>$
x:OV:=lvar.1
ldeg:=degree(ffactor,lvar)
   --- Is the polynomial linear in one of the variables ? ---
member?(1,ldeg) =>
x:OV:=lvar.position(1,ldeg)
lcont:= gcd coefficients(univariate(ffactor,x))
ffactor:=(ffactor exquo lcont)::P
factorlist:=cons([ffactor,ffexp]$MParFact,factorlist)
for lcterm in mFactor(lcont,dx).factors repeat
   factorlist:=cons([lcterm.irr,lcterm.pow * ffexp], factorlist)
varch:=varChoose(ffactor,lvar,ldeg)
um:=varch.npol
ldeg:=ldeg.rest
lvar:=lvar.rest
if varch.nvar.1 ^= x then
   lvar:= varch.nvar
   x := lvar.1
   lvar:=lvar.rest
   pc:= gcd coefficients um
   if pc^=1 then
      um:=(um exquo pc)::SUP P
      ffactor:=multivariate(um,x)
      for lcterm in mFactor(pc,dx).factors repeat
         factorlist:=cons([lcterm.irr,lcterm.pow*ffexp],factorlist)
ldeg:= degree(ffactor,lvar)
   -- should be unitNormal if unified, but for now it is easier
   lcum:F:= leadingCoefficient leadingCoefficient
      leadingCoefficient um
   if lcum ^=1 then
      um:=((inv lcum)::R::P) * um
      flead.contp := (lcum::R) *flead.contp
      if ground?(leadingCoefficient um)
      then lf:= mfconst(um,dx,lvar,ldeg)
      else lf:=mfpol(um,dx,lvar,ldeg)
auxfl:=[[multivariate(lfp,x),ffexp]$MParFact for lfp in lf]
factorlist:=append(factorlist,auxfl)
flead.factors:= factorlist
flead

pM(lum:L SUP R) : R ==
x := monomial(1,1)$R
for i in 1 .. size()$F repeat
   p := x + (index(i::PositiveInteger)$F) ::R
testModulus(p,lum) => return p
for e in 2.. repeat
    p := (createIrreduciblePoly(e::PositiveInteger))$FFPOLY
    testModulus(p, lum) => return p
    while not((q := nextIrreduciblePoly(p)$FFPOLY) case "failed") repeat
        p := q::SUP F
        if testModulus(p, lum)$GenExEuclid(R, SUP R) then return p

---- push x in the coefficient domain for a term ----
pushdcoef(t:PG):P ==
    map((f1:F):R+->coerce(f1)$R,t)$MPolyCatFunctions2(OV,E,
        IndexedExponents OV,F,R,PG,P)

---- internal function, for testing bad cases ----
intfact(um:SUP P,lvar: L OV,ldeg:L NNI,
    tleadpol:MFinalFact,ltry:L L R): L SUP P ==
    polcase:Boolean:=(not empty? tleadpol.factors )
    vfchoo:Valuf:=
        polcase =>
            leadpol:L P:=[ff.irr for ff in tleadpol.factors]
            intChoose(um,lvar,tleadpol.contp,leadpol,ltry)
            intChoose(um,lvar,1,empty(),empty())
        unifact:List SUP R := vfchoo.unvfact
        nfact:NNI := #unifact
        nfact=1 => [um]
    ltry:L L R:= vfchoo.inval
    lval:L R:=first ltry
    dd:= vfchoo.lu
    lpol:List P:=empty()
    leadval:List R:=empty()
    if polcase then
        leadval := vfchoo.complead
        distf := distFact(vfchoo.lu,unifact,tleadpol,leadval,lvar,lval)
        distf case "failed" =>
            return intfact(um,lvar,ldeg,tleadpol,ltry)
        dist := distf :: LeadFact
        -- check the factorization of leading coefficient
        lpol:= dist.polfac
        dd := dist.correct
        unifact:=dist.corrfact
        if dd^=1 then
            unifact := [dd*unifact.i for i in 1..nfact]
            um := ((dd**((nfact-1)::NNI)):P)*um
            (ffin:= lifting(um,lvar,unifact,lval,lpol,1deg,pM(unifact)))
            case "failed" => intfact(um,lvar,1deg,tleadpol,ltry)
        factfin: L SUP P:=ffin :: L SUP P
        if dd^=1 then
            factfin:=[primitivePart ff for ff in factfin]
factfin
-- the following functions are used to "push" x in the coefficient ring --
---- push back the variable ----
pushup(f:P,x:OV) : PG ==
  ground? f => pushupconst((retract f)@R,x)
  rr:PG:=0
  while f^=0 repeat
    lf:=leadingMonomial f
    cf:=pushupconst(leadingCoefficient f,x)
    lvf:=variables lf
    rr:=rr+monomial(cf,lvf, degree(lf,lvf))$PG
    f:=reductum f
  rr

---- push x in the coefficient domain for a polynomial ----
pushdown(g:PG,x:OV) : P ==
  ground? g => ((retract g)@F)::R::P
  rf:P:=0$P
  ug:=univariate(g,x)
  while ug^=0 repeat
    cf:=monomial(1,degree ug)$R
    rf:=rf+cf*pushdcoef(leadingCoefficient ug)
    ug := reductum ug
  rf

---- push x back from the coefficient domain ----
pushupconst(r:R,x:OV):PG ==
  ground? r => (retract r)@F ::PG
  rr:PG:=0
  while r^=0 repeat
    rr:=rr+monomial((leadingCoefficient r)::PG,x,degree r)$PG
    r:=reductum r
  rr

-- This function has to be added to Eucliden domain
ran(k1:Z) : R ==
  --if R case Integer then random()$R rem (2*k1)-k1
  --else
  +/[monomial(random()$F,i)$R for i in 0..k1]

checkzero(u:SUP P,um:SUP R) : Boolean ==
  u=0 => um =0
  um = 0 => false
  degree u = degree um => checkzero(reductum u, reductum um)
  false

  --- Choose the variable of least degree ---
varChoose(m:P,lvar:L OV,ldeg:L NNI) : NewOrd ==
  k:="min"/[d for d in ldeg]
  k=degree(m,first lvar) =>
    [univariate(m,first lvar),lvar,ldeg]$NewOrd
i := position(k, ldeg)
x:OV := lvar.i
ldeg := cons(k, delete(ldeg, i))
lvar := cons(x, delete(lvar, i))
\[\text{univariate}(m, x), lvar, ldeg\]$ NewOrd

\[\text{norm}(\text{lum}: \text{L SUP R}): \text{Integer} = "\text{max"}/[\text{degree lup for lup in lum}]\]

--- Choose the values to reduce to the univariate case ---

\(\text{intChoose}(\text{um}: \text{SUP P}, \text{lvar}: \text{L OV}, \text{clc}: \text{R}, \text{plist}: \text{L P}, \text{ltry}: \text{L L R}) : \text{Valuf} =\)

-- declarations

degum:NNI := degree um
nvar1 := #lvar
range:NNI := 0
unifact: L SUP R
ctf1: R := 1
testp:Boolean := -- polynomial leading coefficient
    plist = empty() => false
    true
leadcomp, leadcomp1 : L R
leadcomp := leadcomp1 := empty()
nfatt:NNI := degum +1
lffc:R := 1
lffc1 := lffc
newunifact : L SUP R := empty()
leadtest := true --- the lc test with polCase has to be performed
int:L R := empty()

-- New sets of values are chosen until we find twice the
-- same number of "univariate" factors:the set smaller in modulo is
-- is chosen.

while true repeat
    lval := [ran(range) for i in 1..nvar1]
    member?(lval, ltry) => range := i + range
    ltry := cons(lval, ltry)
    leadcomp1 := [retract eval(pol, lvar, lval) for pol in plist]
    testp and or/[unit? epl for epl in leadcomp1] => range := range + 1
    newm: SUP R := completeEval(um, lvar, lval)
    degum = degree newm or minimumDegree newm ^= 0 => range := range + 1
    lffc1 := content newm
    newm := (newm exquo lffc1)::$SUP R
    testp and leadtest and ~ polCase(lffc1*clc, #plist, leadcomp1)
        => range := range + 1
    Dnewm := differentiate newm
    D2newm := map(differentiate, newm)
    degree(gcd \[newm, Dnewm, D2newm]\) ^= 0 => range := range + 1
    -- if R has Integer then luniv := henselFact(newm, false)$
    -- else
    lcnm:F := 1
-- should be unitNormal if unified, but for now it is easier
if (lcnm:=leadingCoefficient newm)^=1 then
  newm:=(inv lcnm)::R*newm
dx:="max"/degree uc for uc in coefficients newm]
luniv:=generalTwoFactor(newm)$TwoFactorize(F)
lunivf:= factors luniv

nf:= #lunivf

nf=0 or nf>nfatt => "next values" --- pretest failed ---

--- the univariate polynomial is irreducible ---
if nf=1 then leave (unifact:=[newm])

lffc1:=lcnm * retract(unit luniv)@R * lffc1

-- the new integer give the same number of factors
nfatt = nf =>
  -- if this is the first univariate factorization with polCase=true
  -- or if the last factorization has smaller norm and satisfies
  -- polCase
  if leadtest or
    ((norm unifact > norm [ff.factor for ff in lunivf]) and
     (^testp or polCase(lffc1*clc,#plist,leadcomp1))) then
      unifact:=[uf.factor for uf in lunivf]
      int:=lval
      lffc:=lffc1
      if testp then leadcomp:=leadcomp1
      leave "foundit"

-- the first univariate factorization, initialize
nfatt > degum =>
  unifact:=[uf.factor for uf in lunivf]
  lffc:=lffc1
  if testp then leadcomp:=leadcomp1
  int:=lval
  leadtest := false
  nfatt := nf

nfatt<nfatt => -- for the previous values there were more factors
  if testp then leadtest:="polCase(lffc*clc,#plist,leadcomp)"
  else leadtest:= false

  -- if polCase=true we can consider the univariate decomposition
  if ^leadtest then
    unifact:=[uf.factor for uf in lunivf]
    lffc:=lffc1
    if testp then leadcomp:=leadcomp1
    int:=lval
    nfatt := nf

[cons(int,ltry),unifact,lffc,leadcomp]$Valuf
constantCase(m:P,factorlist:List MParFact) : MFinalFact ==
--if R case Integer then [const m,factorlist]$MFinalFact
--else
lumn:=distdfact((retract m)@R,false)$DistinctDegreeFactorize(F,R)
[(lumn.cont)::R, append(factorlist,
[(pp.irr)::P,pp.pow] for pp in lumn.factors)]$MFinalFact

---- The polynomial has mindeg>0 ----
simplify(m:P,dm:Z,lvar:LOV,lmdeg:LNNI):MFinalFact ==
factorlist:L MParFact:=empty()
pol1:P:= 1$P
for x in lvar repeat
  i := lmdeg.(position(x,lvar))
i=0 => "next value"
pol1:=pol1*monomial(1$P,x,i)
factorlist:=cons([x::P,i]$MParFact,factorlist)
m := (m exquo pol1)::P
ground? m => constantCase(m,factorlist)
flead:=mFactor(m,dm)
flead=factors:=append(factorlist,flead.factors)
flead

---- m square-free,primitive,lc constant ----
nsign:Boolean
factfin:L SUP P:=empty()
empty? lvar =>
  um1:SUP R:=map(ground,
    um)$UPCF2(P,SUP P,R,SUP R)
lum:= generalTwoFactor(um1)$TwoFactorize(F)
  [map(coerce,lumf.factor)$UPCF2(R,SUP R,P,SUP P)
   for lumf in factors lum]
intfact(um,lvar,ldeg,[0,empty()])$MFinalFact,empty())

--- m is square-free,primitive,lc is a polynomial ---
dist : LeadFact
tleadpol:=mFactor(leadingCoefficient um,dm)
intfact(um,lvar,ldeg,tleadpol,empty())

factor(m:PG):Factored PG ==
lv:=variables m
lv=empty() => makeFR(m,empty() )
-- reduce to multivariate over SUP
ld:=[degree(m,x) for x in lv]
dx:="min"/ld
basicVar:=lv(position(dx,ld))
cm:=pushdown(m,basicVar)
flist := mFactor(cm,dx)
pushupconst(flist.contp,basicVar) *
  (*/[primeFactor(pushup(u.irr,basicVar),u.pow)
    for u in flist.factors])
See Also:
  o )show MultipleMap

---

**MultipleMap (MMAP)**

Exports:
map

--- package MMAP MultipleMap ---

)abbrev package MMAP MultipleMap
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
  ++ Lifting of a map through 2 levels of polynomials;

MultipleMap(R1,UP1,UPUP1,R2,UP2,UPUP2): Exports == Implementation where
  R1 : IntegralDomain
  UP1 : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  R2 : IntegralDomain
  UP2 : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2

  Q1 ==> Fraction UP1
  Q2 ==> Fraction UP2

Exports ==> with
  map: (R1 -> R2, UPUP1) -> UPUP2
    ++ map(f, p) lifts f to the domain of p then applies it to p.
Implementation ==> add
   import UnivariatePolynomialCategoryFunctions2(R1, UP1, R2, UP2)

rfmap: (R1 -> R2, Q1) -> Q2
rfmap(f, q) == map(f, numer q) / map(f, denom q)

map(f, p) ==
   map(x +-> rfmap(f,x),
       p)$UnivariatePolynomialCategoryFunctions2(Q1, UPUP1, Q2, UPUP2)

— MMAP.dotabb —

"MMAP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MMAP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MMAP" -> "PFECAT"

package MCALCFN MultiVariableCalculusFunctions

— MultiVariableCalculusFunctions.input —

)set break resume
)sys rm -f MultiVariableCalculusFunctions.output
)spool MultiVariableCalculusFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultiVariableCalculusFunctions
--E 1

)spool
)lisp (bye)

— MultiVariableCalculusFunctions.help —
MultiVariableCalculusFunctions Package provides several functions for multivariable calculus. These include gradient, hessian and jacobian, divergence and laplacian. Various forms for banded and sparse storage of matrices are included.

See Also:
- )show MultiVariableCalculusFunctions

---

MultiVariableCalculusFunctions (MCALCFN)

Exports:
- bandedHessian
- bandedJacobian
- divergence
- gradient
- hessian
- jacobian
- laplacian

--- package MCALCFN MultiVariableCalculusFunctions ---

)abbrev package MCALCFN MultiVariableCalculusFunctions
++ Author: Themos Tsikas, Grant Keady
++ Date Created: December 1992
++ Date Last Updated: June 1993
++ Description:
++ \spadtype{MultiVariableCalculusFunctions} Package provides several
++ functions for multivariable calculus.
++ These include gradient, hessian and jacobian, divergence and laplacian.
++ Various forms for banded and sparse storage of matrices are included.

MultiVariableCalculusFunctions(S,F,FLAF,FLAS) : Exports == Implementation where
PI => PositiveInteger
NNI ==> NonNegativeInteger

S: SetCategory
F: PartialDifferentialRing(S)
FLAS: FiniteLinearAggregate(S)
    with finiteAggregate
FLAF: FiniteLinearAggregate(F)

Exports ==> with
    gradient: (F,FLAS) -> Vector F
       \spad{gradient(v,xlist)}
       computes the gradient, the vector of first partial derivatives,
       of the scalar field v,
       v a function of the variables listed in xlist.
    divergence: (FLAF,FLAS) -> F
       \spad{divergence(vf,xlist)}
       computes the divergence of the vector field vf,
       vf a vector function of the variables listed in xlist.
    laplacian: (F,FLAS) -> F
       \spad{laplacian(v,xlist)}
       computes the laplacian of the scalar field v,
       v a function of the variables listed in xlist.
    hessian: (F,FLAS) -> Matrix F
       \spad{hessian(v,xlist)}
       computes the hessian, the matrix of second partial derivatives,
       of the scalar field v,
       v a function of the variables listed in xlist.
    bandedHessian: (F,FLAS,NNI) -> Matrix F
       \spad{bandedHessian(v,xlist,k)}
       computes the hessian, the matrix of second partial derivatives,
       of the scalar field v,
       v a function of the variables listed in xlist,
       k is the semi-bandwidth, the number of nonzero subdiagonals,
       2*k+1 being actual bandwidth.
       Stores the nonzero band in lower triangle in a matrix,
       dimensions k+1 by #xlist,
       whose rows are the vectors formed by diagonal, subdiagonal, etc.
       of the real, full-matrix, hessian.
       (The notation conforms to LAPACK/NAG-F07 conventions.)
-- At one stage it seemed a good idea to help the ASP<n> domains
-- with the types of their input arguments and this led to the
-- standard Gradient|Hessian|Jacobian functions.
-- standardJacobian: (Vector(F),List(S)) -> Matrix F
-- \spad{jacobian(vf,xlist)}
-- computes the jacobian, the matrix of first partial derivatives,
-- of the vector field vf,
-- vf a vector function of the variables listed in xlist.
    jacobian: (FLAF,FLAS) -> Matrix F
       \spad{jacobian(vf,xlist)}
       computes the jacobian, the matrix of first partial derivatives,
++ of the vector field \( \mathbf{v} \),
++ \( \mathbf{v} \) a vector function of the variables listed in \( \mathbf{x} \).
\[ \text{bandedJacobian}: (\text{FLAF}, \text{FLAS}, \text{NNI}, \text{NNI}) \to \text{Matrix F} \]
++ \( \text{\\texttt{bandedJacobian}}(\mathbf{v}, \mathbf{x}, \mathbf{k}, \mathbf{k}) \)
++ computes the Jacobian, the matrix of first partial derivatives,
++ of the vector field \( \mathbf{v} \),
++ \( \mathbf{v} \) a vector function of the variables listed in \( \mathbf{x} \),
++ \( \mathbf{k} \) is the number of nonzero subdiagonals,
++ \( \mathbf{k} \) is the number of nonzero superdiagonals,
++ \( \mathbf{k}+\mathbf{k}+1 \) being actual bandwidth.
++ Stores the nonzero band in a matrix,
++ dimensions \( \mathbf{k}+\mathbf{k}+1 \) by \( \#\mathbf{x} \).
++ The upper triangle is in the top \( \mathbf{k} \) rows,
++ the diagonal is in row \( \mathbf{k}+1 \),
++ the lower triangle in the last \( \mathbf{k} \) rows.
++ Entries in a column in the band store correspond to entries
++ in same column of full store.
++ (The notation conforms to LAPACK/NAG-F07 conventions.)

Implementation ==> add
localGradient\( (\mathbf{v}: \text{F}, \mathbf{x}: \text{List(S)}) : \text{Vector(F)} \) ==
\( \text{vector}([D(\mathbf{v}, x) \text{ for } x \text{ in } \mathbf{x}]) \)
\[ \text{gradient}(\mathbf{v}, \mathbf{x}) = \]
\( \text{--xlist:List(S)} := [\mathbf{x}(i) \text{ for } i \text{ in } 1 \ldots \text{maxIndex(xflas)}] \)
\( \text{xlist:List(S)} := \text{parts(xflas)} \)
\( \text{localGradient}(\mathbf{v}, \mathbf{xlist}) \)

localDivergence\( (\mathbf{vf}: \text{Vector(F)}, \mathbf{x}: \text{List(S)}) : \text{F} \) ==
\( i: \text{PI} \)
\( n: \text{NNI} \)
\( \text{ans}: \text{F} \)
\( -- \text{Perhaps should report error if two args of min different} \)
\( n:=\text{min(}(\#(\mathbf{xlist}), ((\text{maxIndex(vf)}):\text{NNI}))\text{NNI}) \)
\( \text{ans}:=0 \)
\( \text{for } i \text{ in } 1 \ldots n \text{ repeat } \text{ans} := \text{ans} + D(\mathbf{vf}(i), \mathbf{xlist}(i)) \)
\( \text{ans} \)

divergence\( (\mathbf{vf}, \mathbf{x}) = \)
\( \text{xlist:List(S)} := \text{parts(xflas)} \)
\( i: \text{PI} \)
\( n: \text{NNI} \)
\( \text{ans}: \text{F} \)
\( -- \text{Perhaps should report error if two args of min different} \)
\( n:=\text{min(}(\#(\mathbf{xlist}), ((\text{maxIndex(vf)}):\text{NNI}))\text{NNI}) \)
\( \text{ans}:=0 \)
\( \text{for } i \text{ in } 1 \ldots n \text{ repeat } \text{ans} := \text{ans} + D(\mathbf{vf}(i), \mathbf{xlist}(i)) \)
\( \text{ans} \)

laplacian\( (\mathbf{v}, \mathbf{x}) = \)
\( \text{xlist:List(S)} := \text{parts(xflas)} \)
\( \text{gv:Vector(F)} := \text{localGradient}(\mathbf{v}, \mathbf{xlist}) \)
\( \text{localDivergence(gv, xlist)} \)
hessian\( (\mathbf{v}, \mathbf{x}) = \)
xlist:List(S) := parts(xflas)
matrix([[D(v,[x,y]) for x in xlist] for y in xlist])
--standardJacobian(vf,xlist) ==
--  i: PI
--  matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
jacobian(vf,xflas) ==
xlist:List(S) := parts(xflas)
i: PI
matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
bandedHessain(vf,xflas,k) ==
xlist:List(S) := parts(xflas)
j,iw: PI
n: NNI
bandM: Matrix F
n:= #(xlist)
bandM:= new(k+1,n,0)
for j in 1 .. n repeat setelt(bandM,1,j,D(v,xlist(j),2))
for iw in 2 .. (k+1) repeat (_
  for j in 1 .. (n-iw+1) repeat (_
    setelt(bandM,iw,j,D(v,[xlist(j),xlist(j+iw-1)])) ) )
bandM

jacobian(vf,xflas) ==
xlist:List(S) := parts(xflas)
i: PI
matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
bandedJacobian(vf,xflas,kl,ku) ==
xlist:List(S) := parts(xflas)
j,iw: PI
n: NNI
bandM: Matrix F
n:= #(xlist)
bandM:= new(kl+ku+1,n,0)
for j in 1 .. n repeat setelt(bandM,ku+1,j,D(vf(j),xlist(j)))
for iw in (ku+2) .. (ku+kl+1) repeat (_
  for j in 1 .. (n-iw+ku+1) repeat (_
    setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j))) ) )
for iw in 1 .. ku repeat (_
  for j in (ku+2-iw) .. n repeat (_
    setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j))) ) )
bandM

—— MCALCFN.dotabb ——

"MCALCFN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MCALCFN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"MCALCFN" -> "IVECTOR"
package MULTFACT MultivariateFactorize

— MultivariateFactorize.input —

)set break resume
)sys rm -f MultivariateFactorize.output
)spool MultivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultivariateFactorize
--E 1

)spool
)lisp (bye)

— MultivariateFactorize.help —

====================================================================
MultivariateFactorize examples
====================================================================

This is the top level package for doing multivariate factorization over basic domains like Integer or Fraction Integer.

See Also:
  o )show MultivariateFactorize
MultivariateFactorize (MULTFACT)

Exports:
factor

--- package MULTFACT MultivariateFactorize ---

)abbrev package MULTFACT MultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \spadtype{Integer} or \spadtype{Fraction Integer}.

MultivariateFactorize(OV,E,R,P) : C == T
where
  R : Join(EuclideanDomain, CharacteristicZero)
    -- with factor on R[x]
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)
  Z ==> Integer
  MParFact ==> Record(irr:P,pow:Z)
  USP ==> SparseUnivariatePolynomial P
  SUParFact ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact ==> Record(contp:R,factors:List MParFact)

          -- contp = content,
          -- factors = List of irreducible factors with exponent
  L ==> List

C == with
  factor : P -> Factored P
    ++ factor(p) factors the multivariate polynomial p over its coefficient
    ++ domain
factor : USP -> Factored USP
++ factor(p) factors the multivariate polynomial p over its coefficient
++ domain where p is represented as a univariate polynomial with
++ multivariate coefficients
T == add
factor(p:P) : Factored P ==
  R is Fraction Integer =>
    factor(p)$MRationalFactorize(E,OV,Integer,P)
  R is Fraction Complex Integer =>
    factor(p)$MRationalFactorize(E,OV,Complex Integer,P)
  R is Fraction Polynomial Integer and OV has convert: % -> Symbol =>
    factor(p)$MPolyCatRationalFunctionFactorizer(E,OV,Integer,P)
  factor(p,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)

factor(up:USP) : Factored USP ==
  factor(up,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)

package MLIFT MultivariateLifting

— MultivariateLifting.input —

)set break resume
)sys rm -f MultivariateLifting.output
)spool MultivariateLifting.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show MultivariateLifting
--E 1

)spool
)lisp (bye)
---

MultivariateLifting.help

====================================================================
MultivariateLifting examples
====================================================================

See Also:
  o )show MultivariateLifting

/package MLIFT MultivariateLifting |

)abbrev package MLIFT MultivariateLifting
++ Author : P.Gianni.
++ Description:
++ This package provides the functions for the multivariate "lifting", using
++ an algorithm of Paul Wang.
++ This package will work for every euclidean domain R which has property
++ F, i.e. there exists a factor operation in \spad{R[x]}.

MultivariateLifting(E,OV,R,P) : C == T
where
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  R : EuclideanDomain -- with property "F"
  Z ==> Integer
  BP ==> SparseUnivariatePolynomial R
  P : PolynomialCategory(R,E,OV)
  SUP ==> SparseUnivariatePolynomial P
  NNI ==> NonNegativeInteger
  Term ==> Record(expt:NNI,pcoef:P)
  VTerm ==> List Term
  Table ==> Vector List BP
  L ==> List

  C == with
    corrPoly: (SUP,L OV,L R,L NNI,L SUP,Table,R) -> Union(L SUP,"failed")
++ corrPoly(u,lv,lr,ln,lu,t,r) \ undocumented
lifting: (SUP,OV,L BP,L R,P,L NNI,R) -> Union(L SUP,"failed")
++ lifting(u,lv,lu,lr,lp,ln,r) \ undocumented
lifting1: (SUP,OV,L SUP,L R,P,L VTerm,L NNI,Table,R) ->
++ lifting1(u,lv,lu,lr,lp,lt,ln,t,r) \ undocumented

T == add
GenExEuclid(R,BP)
NPCoef(BP,E,OV,R,P)
IntegerCombinatoricFunctions(Z)

SUPF2 ==> SparseUnivariatePolynomialFunctions2

DetCoef ==> Record(deter:L SUP,dterm:L VTerm,nfacts:L BP,
nlead:L P)

--- local functions ---
normalDerivM : (P,Z,OV) -> P
normalDeriv : (SUP,Z) -> SUP
subslead : (SUP,P) -> SUP
subscoef : (SUP,L Term) -> SUP
maxDegree : (SUP,OV) -> NonNegativeInteger

corrPoly(m:SUP,lvar:OV,fval:L R,ld:L NNI,flist:L SUP,
table:Table,pmod:R):Union(L SUP,"failed") ==
-- The correction coefficients are evaluated recursively.
-- Extended Euclidean algorithm for the multivariate case.
-- the polynomial is univariate --
#lvar=0 =>
lp:=solveid(map(ground,m)$SUPF2(P,R),pmod,table)
if lp case "failed" then return "failed"
lcoef:= [map(coerce,mp)$SUPF2(R,P) for mp in lp::L BP]

diff,ddiff,pol,polc:SUP
listpolv,listcong:L SUP
deg1:NNI:= ld.first
np:NNI:= #flist
a:P:= fval.first ::P
y:OV:=lvar.first
lvar:=lvar.rest
listpolv:L SUP := [map((p1:P):P +-> eval(p1,y,a),f1) for f1 in flist]
u1:=map((p1:P):P +-> eval(p1,y,a),m)
flcoef:=corrPoly(u1,lvar,fval.rest,ld.rest,listpolv,table,pmod)
if flcoef case "failed" then return "failed"
else lcoef:=flcoef :: L SUP
listcong:=[/*[flist.i for i in 1..np | i^=1] for l in 1..np]
polc := (monomial(1, y, 1) - a)

for l in 1..deg1 repeat
  if diff=0 then return lcoef
  pol := pol*polc
  (ddiff := map((p:P):P+->eval(normalDerivM(p, l, y, a), diff)) = 0
    => "next l"
  fbeta := corrPoly(ddiff, lvar, fval.rest, ld.rest, listpolv, table, pmod)
  if fbeta case "failed" then return "failed"
  else beta := fbeta :: L SUP
    lcoef := [lcoef.i + beta.i*pol for i in 1..np]
    diff := diff - +/[listcong.i*beta.i for i in 1..np]*pol
  polc := empty()
  pol := 1$SUP

lifting1(m:SUP, lvar:L OV, plist:L SUP, vlist:L R, tlist:L P, coeflist:L VTerm, listdeg:L NNI, table:Table, pmod:R) : Union(L SUP, "failed") ==
  -- The factors of m (multivariate) are determined,
  -- We suppose to know the true univariate factors
  -- some coefficients are determined
  conglist:L SUP := empty()
  nvar : NNI := #lvar
  pol, polc := empty()
  testp := (not empty?(tlist))
  lalpha := empty()
  tlv := empty()
  subsvar := empty()
  subsval := empty()
  li := lvar
  ldeg := empty()
  clv := empty()
  -- for j in 1..nvar repeat
  for j in 1..nvar repeat
    x := li.first
    li := rest li
    conglist := plist
    v := vlist.first
    vlist := rest vlist
    degj := listdeg.j
    ldeg := cons(degj, ldeg)
    subsvar := cons(x, subsvar)
    subsval := cons(v, subsval)
    if testp then
      for j in 1..nvar repeat
        tlv := [eval(p, li, vlist) for p in tlist]
        clv := [[term.expt, eval(term.pcoef, li, vlist)]$Term
                    for term in clist] for clist in coeflist]
else (tlv,clv):=(tlist,coeflist)
plist :=\[subslead(p,lcp) for p in plist for lcp in tlv\]
if not(empty? coeflist) then
   plist:=\[subscoef(tpol,clist)
   for tpol in plist for clist in clv\]
mj := map((p1:P):P+->eval(p1,li,vlist),m) --m(x1,,xj,aj+1,,an
polc := x::P - v::P --(xj-aj)
pol:= 1$P
--Construction of Rik, k in 1..right degree for xj+1
for k in 1..degj repeat --I can exit before
   pol := pol*polc
   mc := */[term for term in plist]-mj
   if mc=0 then leave "next var"
   --Modulus Dk
   mc:=map((p1:P):P +-> normalDerivM(p1,k,x),mc)
   (mc := map((p1:P):P +-> eval(p1,[x],[v]),mc))=0 => "next k"
   flalpha:=corrPoly(mc,subsvar.rest,subsval.rest,
   ldeg.rest,conglist,table,pmod)
   if flalpha case "failed" then return "failed"
   else lalpaha:=flalpha :: L SUP
   plist:=\[term-alpha*pol for term in plist for alpha in lalpaha\]
   -- PGCD may call with a smaller valure of degj
   idegj:Integer:=maxDegree(m,x)
   for term in plist repeat idegj:=idegj -maxDegree(term,x)
   idegj < 0 => return "failed"
plist
   --There are not extraneous factors
maxDegree(um:SUP,x:OV):NonNegativeInteger ==
   ans:NonNegativeInteger:=0
   while um ^= 0 repeat
      ans:=max(ans,degree(leadingCoefficient um,x))
      um:=reductum um
   ans
lifting(um:SUP,lvar:L OV,plist:L BP,vlist:L R,
   tlist:L P,listdeg:L NNI,pmod:R):Union(L SUP,"failed") ==
-- The factors of m (multivariate) are determined, when the
-- univariate true factors are known and some coefficient determined
nplist:List SUP:=[map(coerce,pp)$SUPF2(R,P) for pp in plist]
empty? tlist =>
   table:=tablePow(degree um,pmod,plist)
   table case "failed" => error "Table construction failed in MLIFT"
   lifting1(um,lvar,nplist,vlist,tlist,empty(),listdeg,table,pmod)
ldcoef:DetCoef:=npcoef(um,plist,tlist)
if not empty?(listdet:=ldcoef.deter) then
   if #listdet = #plist then return listdet
   plist:=ldcoef.nfacts
   nplist:=[map(coerce,pp)$SUPF2(R,P) for pp in plist]
   um:=(um exquo */[pol for pol in listdet]):SUP
tlist:=ldcoef.nlead
    tab:=tablePow(degree um,pmod,plist.rest)
else tab:=tablePow(degree um,pmod,plist)
tab case "failed" => error "Table construction failed in MLIFT"
table:Table:=tab
ffl:=lifting1(um,lvar,nplist,vlist,tlist,ldcoef.dterm,listdeg,table,pmod)
if ffl case "failed" then return "failed"
append(listdet,ffl:: L SUP)

-- normalDerivM(f,m,x) = the normalized (divided by m!) m-th
-- derivative with respect to x of the multivariate polynomial f
normalDerivM(g:P,m:Z,x:OV) : P ==
multivariate(normalDeriv(univariate(g,x),m),x)

normalDeriv(f: SUP,m: Z) : SUP ==
  (n1:Z:=degree f) < m => 0$SUP
  n1=m => leadingCoefficient f :: SUP
  k:=binomial(n1,m)
  ris: SUP := 0$SUP
  n:=n1
  while n>= m repeat
    while n1>n repeat
      k:=(k*(n1-m)) quo n1
      n1:=n1-1
      ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
      f:=reductum f
      n:=degree f
    ris
subsead(m:SUP,pol:P):SUP ==
  dm:NNI:=degree m
  monomial(pol,dm)+reductum m

subseof(um:SUP,lterm:L Term):SUP ==
  dm:NNI:=degree um
  new:=monomial(leadingCoefficient um,dm)
  for k in dm-1..0 by -1 repeat
    i:NNI:=k::NNI
    empty?(lterm) or lterm.first.expt^=i =>
      new:=new+monomial(coefficient(um,i),i)
      new:=new+monomial(lterm.first.pcoef,i)
lterm:=lterm.rest
new
package MULTSQFR MultivariateSquareFree

— MultivariateSquareFree.input —

)set break resume
/sys rm -f MultivariateSquareFree.output
/spool MultivariateSquareFree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultivariateSquareFree
--E 1

)spool
)lisp (bye)

— MultivariateSquareFree.help —

====================================================================
MultivariateSquareFree examples
====================================================================

This package provides the functions for the computation of the square free decomposition of a multivariate polynomial. It uses the package GenExEuclid for the resolution of the equation $Af + Bg = h$ and its generalization to $n$ polynomials over an integral domain and the package MultivariateLifting for the "multivariate" lifting.

See Also:
  o )show MultivariateSquareFree

——
MultivariateSquareFree (MULTSQFR)

Exports:
check coefChoose compdegd consnewpol intChoose
lift myDegree normDeriv2 nsqfree squareFree
squareFreePrim univcase

— package MULTSQFR MultivariateSquareFree —

)abbrev package MULTSQFR MultivariateSquareFree
++ Author : P.Gianni
++ Description:
++ This package provides the functions for the computation of the square
++ free decomposition of a multivariate polynomial.
++ It uses the package GenExEuclid for the resolution of
++ the equation \( \spad{Af + Bg = h} \) and its generalization to \( n \) polynomials
++ over an integral domain and the package \spad{MultivariateLifting}
++ for the "multivariate" lifting.
MultivariateSquareFree (E,OV,R,P) : C == T where
Z == Integer
NNI == NonNegativeInteger
R : EuclideanDomain
OV : OrderedSet
E : OrderedAbelianMonoidSup
P : PolynomialCategory(R,E,OV)
SUP ==> SparseUnivariatePolynomial P

BP ==> SparseUnivariatePolynomial(R)
fUnion ==> Union("nil","sqfr","irred","prime")
ffSUP ==> Record(flg:fUnion,fctr:SUP,xpnt:Integer)
ffP ==> Record(flg:fUnion,fctr:P,xpnt:Integer)
FFE ==> Record(factor:BP,exponent:Z)
FFEP ==> Record(factor:P,exponent:Z)
FFES ==> Record(factor:SUP,exponent:Z)
Choice ==> Record(upol:BP,Lval:List(R),Lfact:List FFE,ctpol:R)
squareForm ==> Record(unitPart:P,suPart:List FFES)
Twopol ==> Record(pol: SUP, polval: BP)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2

C == with
  squareFree : P -> Factored P
  ++ squareFree(p) computes the square free
  ++ decomposition of a multivariate polynomial p.
  squareFree : SUP -> Factored SUP
  ++ squareFree(p) computes the square free
  ++ decomposition of a multivariate polynomial p presented as
  ++ a univariate polynomial with multivariate coefficients.
  squareFreePrim : P -> Factored P
  ++ squareFreePrim(p) compute the square free decomposition
  ++ of a primitive multivariate polynomial p.

  ---- local functions ----
  compdegd : List FFE -> Z
  ++ compdegd should be local
  univcase : (P, OV) -> Factored(P)
  ++ univcase should be local
  consnewpol : (SUP, BP, Z) -> Twopol
  ++ consnewpol should be local
  nsqfree : (SUP, List(OV), List List R) -> squareForm
  ++ nsqfree should be local
  intChoose : (SUP, List(OV), List List R) -> Choice
  ++ intChoose should be local
  coefChoose : (Z, Factored P) -> P
  ++ coefChoose should be local
  check : (List(FFE), List(FFE)) -> Boolean
  ++ check should be local
  lift : (SUP, BP, BP, P, List(OV), List(NNI), List(R)) -> Union(List(SUP), "failed")
  ++ lift should be local
  myDegree : (SUP, List OV, NNI) -> List NNI
  ++ myDegree should be local
  normDeriv2 : (BP, Z) -> BP
  ++ normDeriv2 should be local

T == add

  pmod: R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

import GenExEuclid()
import MultivariateLifting(E, OV, R, P)
import PolynomialGcdPackage(E, OV, R, P)
import FactoringUtilities(E,OV,R,P)
import IntegerCombinatoricFunctions(Z)

---- Are the univariate square-free decompositions consistent? ----

---- new square-free algorithm for primitive polynomial ----
nsqfree(oldf:SUP,lvar:List(OV),ltry:List List R) : squareForm ==
f:=oldf
univPol := intChoose(f,lvar,ltry)
-- debug msg
-- if not empty? ltry then output("ltry =", (ltry::OutputForm))$OutputPackage
f0:=univPol.upol
-- the polynomial is square-free
f0=1$BP => [1$P,[[f,1]$FFES]]$squareForm
lfact:List(FFE):=univPol.Lfact
lval:=univPol.Lval
ctf:=univPol.ctpol
leadpol:Boolean:=false
sqdec:List FFES := empty()
exp0:Z:=0
unitsq:P:=1
lcf:P:=leadingCoefficient f
if ctf=1 then
  f0:=ctf*f0
  f:=(ctf:P)*f
  lcf:=ctf*lcf
sqlead:List FFEP:= empty()
sqlc:Factored P:=1
if lcf=1$P then
  leadpol:=true
  sqlc:=squareFree lcf
  unitsq:=unitsq*(unit sqlc)
  sqlead:= factors sqlc
lfact:=sort((z1:FFE,z2:FFE):Boolean +-> z1.exponent > z2.exponent,lfact)
while lfact=[] repeat
  pfact:=lfact.first
  (g0,exp0):=(pfact.factor,pfact.exponent)
  lfact:=lfact.rest
  lfact=[] and exp0 =1 =>
  f := (f exquo (ctf:P)):SUP
  gg := unitNormal leadingCoefficient f
  sqdec:=cons([gg.associate*f,exp0],sqdec)
  return [gg.unit, sqdec]$squareForm
if ctf=1 then g0:=ctf*g0
npol:=consnewpol(f,f0,exp0)
(d,d0):=(npol.pol,npol.polval)
if leadpol then lcoef:=coefChoose(exp0,sqlc)
else lcoef:=1$P
ldeg:=myDegree(f,lvar,exp0::NNI)
result:=lift(d,g0,(d0 exquo g0)::BP,lcoef,lvar,ldeg,lval)
result case "failed" => return nsqfree(oldf,lvar,ltry)
result0:SUP:= (result::List SUP).1
r1:SUP:=result0**(exp0:NNI)
if (h:=f exquo r1) case "failed" then return nsqfree(oldf,lvar,empty())
sqdec:=cons([result0,exp0],sqdec)
f:=h::SUP
f0:=completeEval(h,lvar,lval)
lcr:P:=leadingCoefficient result0
if leadpol and lcr"=1$P then
  for lpfact in sqlead while lcr"=1 repeat
    ground? lcr =>
    unitsq:=(unitsq exquo lcr)::P
    lcr:=1$P
    (h1:=lcr exquo lpfact.factor) case "failed" => "next"
    lcr:=h1::P
    lpfact.exponent:=(lpfact.exponent)-exp0
    \[\((\text{retract f}) \text{ exquo ctf)}::P,sqdec\]$squareForm

squareFree(f:SUP) : Factored SUP ==
  degree f =0 =>
  fu:=squareFree retract f
  makeFR((unit fu)::SUP,[["sqfr",ff.fctr::SUP,ff.xpnt]
    for ff in factorList fu])
lvar:= "setUnion"/[variables cf for cf in coefficients f]
empty? lvar => -- the polynomial is univariate
  upol:=map(ground,f)$UPCF2(P,SUP,R,BP)
  usqfr:=squareFree upol
  makeFR(map(coerce,unit usqfr)$UPCF2(R,BP,P,SUP),
    [["sqfr",map(coerce,ff.fctr)$UPCF2(R,BP,P,SUP),ff.xpnt]
      for ff in factorList usqfr])
lcf:=content f
f:=(f exquo lcf) ::SUP
lcSq:=squareFree lcf
lfs:List ffSUP:=[["sqfr",ff.fctr ::SUP,ff.xpnt]
  for ff in factorList lcSq]
partSq:=nsqfree(f,lvar,empty())
lfs:=append([[["sqfr",fu.factor,fu.exponent]$ffSUP
    for fu in partSq.suPart],lfs)
  makeFR((unit lcSq * partSq.unitPart) ::SUP,lfs)
squareFree(f:P) : Factored P ==
  ground? f => makeFR(f,[]) --- the polynomial is constant ---
lvar:List(0V):=variables(f)
result1:List ffP:= empty()
PACKAGE MULTISQFR MULTIVARIATESQUAREFREE

lmdeg := minimumDegree(f, lvar)  --- is the mindeg > 0 ? ---
p:P:=qP
for im in 1..#lvar repeat
  (n:=lmdeg.im)=0 => "next im"
y:=lvar.im
p:=p*monomial(qP,y,n)
result1:=cons(['sqfr',y::P,n],result1)
if p^=1$P then
  f := (f exquo f::P)
if ground? f then return makeFR(f, result1)
lvar:=variables(f)

#lvar=1 =>  --- the polynomial is univariate ---
result:=univcase(f,lvar.first)
makeFR(unit result,append(result1,factorList result))

ldeg:=degree(f,lvar)  --- general case ---
m:="min"/[j for j in ldeg|j^=0]
i:Z:=1
for j in ldeg while j>m repeat i:=i+1
x:=lvar.i
lvar:=delete(lvar,i)
f0:=univariate (f,x)
lcont:P:= content f0
nsqfftot:=nsqfree((f0 exquo lcont)::SUP,lvar,empty())
nsqff:List ffP:=['["sqfr",multivariate(fu.factor,x),fu.exponent]']ffP
  for fu in nsqfftot.suPart]
result1:=append(result1,nsqff)
ground? lcont => makeFR(lcont*nsqfftot.unitPart,result1)
sqlead:=squareFree(lcont)
makeFR(unit sqlead*nsqfftot.unitPart,append(result1,factorList sqlead))

-- Choose the integer for the evaluation.  --
-- If the polynomial is square-free the function returns upol=1.  --

intChoose(f:SUP,lvar:List(OV),ltry:List List R):Choice ==
degf:= degree f
try:NNI:=0
nvr:=#lvar
range:Z:=10
lfact1:List(FFE):=[]
lval1:List R := []
lfact:List(FFE)
ctf1:R := 1
f1:BP:=qBP
d1:Z
while range<10000000000 repeat
  range:=2*range
  lval:=[ran(range) for i in 1..nvr]
member?(lval,ltry) => "new integer"
ltry:=cons(lval,ltry)
f0:=completeEval(f,lvar,lval)
degree f0 ^=degf => "new integer"
ctf:=content f0
lfact:List(FFE):=factors(squareFree((f0 exquo (ctf:R)::BP)::BP))

---- the univariate polynomial is square-free ----
if #lfact=1 and (lfact.1).exponent=1 then
  return [1$BP,lval,lfact,1$R]$Choice

d0:=compdegd lfact
    ---- initialize lfact1 ----
try=0 =>
f1:=f0
lfact1:=lfact
ctf1:=ctf
lval1:=lval
d1:=d0
try:=1
d0=d1 =>
  return [f1,lval1,lfact1,ctf1]$Choice

d0 < d1 =>
  try:=1
f1:=f0
lfact1:=lfact
ctf1:=ctf
lval1:=lval
d1:=d0

---- Choose the leading coeff for the lifting ----
coefChoose(exp:Z,sqlead:Factored(P)) : P ==
lcoef:P:=unit(sqlead)
for term in factors(sqlead) repeat
  texp:=term.exponent
  texp<exp => "next term"
  texp=exp => lcoef:=lcoef*term.factor
  lcoef:=lcoef*(term.factor)**((texp quo exp)::NNI)
lcoef

---- Construction of the polynomials for the lifting ----
consnewpol(g:SUP,g0:BP,deg:Z):Twopol ==
deg=1 => [g,g0]$Twopol
deg:=deg-1
[normalDeriv(g,deg),normDeriv2(g0,deg)]$Twopol

---- lift the univariate square-free factor ----
lift(ud:SUP,g0:BP,g1:BP,lcoef:P,lvar:List(OV),
     ldeg:List(NNI),lval:List(R)) : Union(List SUP,"failed") ==
leadpol:Boolean:=false
lcd:P:=leadingCoefficient ud
leadlist:List(P):=empty()

if ~ground?(leadingCoefficient ud) then
    leadpol:=true
    ud:=lcoef*ud
    lcg0:R:=leadingCoefficient g0
    if ground? lcoef then g0:=retract(lcoef) quo lcg0 *g0
    else g0:=(retract(eval(lcoef,lvar,lval)) quo lcg0) * g0
    g1:=lcoef*g1
    leadlist:=[lcoef, lcd]

plist:=lifting(ud,lvar,[g0,g1],lval,leadlist,ldeg,pmod)
plist case "failed" => "failed"

(p0:SUP,p1:SUP):=((plist::List SUP).1,(plist::List SUP).2)
if completeEval(p0,lvar,lval) ^= g0 then (p0,p1):=(p1,p0)
[primitivePart p0,primitivePart p1]

---- the polynomial is univariate ----

univcase(f:P,x:OV) : Factored(P) ==
uf := univariate f
cf:=content uf
uf :=(uf exquo cf)::BP
result:Factored BP:=squareFree uf
makeFR(multivariate(cf*unit result,x),
        [["sqfr",multivariate(term.factor,x),term.exponent]
         for term in factors result])

-- squareFreePrim(p:P) : Factored P ==
-- -- p is content free
-- -- ground? p => makeFR(p,[]) --- the polynomial is constant ---
--
-- lvar:List(OV):=variables p
-- #lvar=1 => --- the polynomial is univariate ---
-- univcase(p,lvar.first)
-- nsqfree(p,lvar,1)

compdegd(lfact:List(FFE)) : Z ==
ris:Z:=0
for pfact in lfact repeat
    ris:=ris+(pfact.exponent -1)*degree pfact.factor
    ris

normDeriv2(f:BP,m:Z) : BP ==
    (n1:Z:=degree f) < m => 0$BP
    n1=m => (leadingCoefficient f)::BP
    k:=binomial(n1,m)
    ris:=0$BP
    n:=n1
    while n>m repeat
while ni>n repeat
  k:=(k*(ni-m)) quo n1
  n1:=n1-1
  ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
  f:=reductum f
  n:=degree f
  ris
ris

myDegree(f:SUP,lvar:List OV,exp:NNI) : List NNI==
  [n quo exp for n in degree(f,lvar)]
Chapter 15

Chapter N

package NAGF02 NagEigenPackage

— NagEigenPackage.input —

)set break resume
)sys rm -f NagEigenPackage.output
)spool NagEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagEigenPackage
--E 1

)spool
)lisp (bye)

— NagEigenPackage.help —

This package uses the NAG Library to compute
* eigenvalues and eigenvectors of a matrix
* eigenvalues and eigenvectors of generalized matrix
* eigenvalue problems
* singular values and singular vectors of a matrix.

F02 -- Eigenvalues and Eigenvectors
Introduction -- F02
Chapter F02

1623
1. Scope of the Chapter

This chapter is concerned with computing

-- eigenvalues and eigenvectors of a matrix

-- eigenvalues and eigenvectors of generalized matrix eigenvalue problems

-- singular values and singular vectors of a matrix.

2. Background to the Problems

2.1. Eigenvalue Problems

In the most usual form of eigenvalue problem we are given a square n by n matrix A and wish to compute \( \lambda \) (an eigenvalue) and \( x \neq 0 \) (an eigenvector) which satisfy the equation

\[ Ax = \lambda x \]

Such problems are called 'standard' eigenvalue problems in contrast to 'generalized' eigenvalue problems where we wish to satisfy the equation

\[ Ax = \lambda Bx \]

B also being a square n by n matrix.

Section 2.1.1 and Section 2.1.2 discuss, respectively, standard and generalized eigenvalue problems where the matrices involved are dense; Section 2.1.3 discusses both types of problem in the case where A and B are sparse (and symmetric).

2.1.1. Standard eigenvalue problems

Some of the routines in this chapter find all the n eigenvalues, some find all the n eigensolutions (eigenvalues and corresponding eigenvectors), and some find a selected group of eigenvalues and/or eigenvectors. The matrix A may be:

(i) general (real or complex)

(ii) real symmetric, or

(iii) complex Hermitian (so that if \( a_{ij} = (\alpha) + i(\beta) \) then \( a_{ij} = (\alpha) - i(\beta) \)).
In all cases the computation starts with a similarity transformation $S^{-1}A = T$, where $S$ is non-singular and is the product of fairly simple matrices, and $T$ has an ‘easier form’ than $A$ so that its eigensolutions are easily determined. The matrices $A$ and $T$, of course, have the same eigenvalues, and if $y$ is an eigenvector of $T$ then $Sy$ is the corresponding eigenvector of $A$.

In case (i) (general real or complex $A$), the selected form of $T$ is an upper Hessenberg matrix ($t_{ij} = 0$ if $i-j > 1$) and $S$ is the product of $n-2$ stabilised elementary transformation matrices. There is no easy method of computing selected eigenvalues of a Hessenberg matrix, so that all eigenvalues are always calculated. In the real case this computation is performed via the Francis QR algorithm with double shifts, and in the complex case by means of the LR algorithm. If the eigenvectors are required they are computed by back-substitution following the QR and LR algorithm.

In case (ii) (real and symmetric $A$) the selected simple form of $T$ is a tridiagonal matrix ($t_{ij} = 0$ if $|i-j| > 1$), and $S$ is the product of $n-2$ orthogonal Householder transformation matrices. If only selected eigenvalues are required, they are obtained by the method of bisection using the Sturm sequence property, and the corresponding eigenvectors of $T$ are computed by inverse iteration. If all eigenvalues are required, they are computed from $T$ via the QL algorithm (an adaptation of the QR algorithm), and the corresponding eigenvectors of $T$ are the product of the transformations for the QL reduction. In all cases the corresponding eigenvectors of $A$ are recovered from the computation of $x = Sy$.

In case (iii) (complex Hermitian $A$) analogous transformations as in case (ii) are used. $T$ has complex elements in off-diagonal positions, but a simple diagonal similarity transformation is then used to produce a real tridiagonal form, after which the QL algorithm and succeeding methods described in the previous paragraph are used to complete the solution.

2.1.2. Generalized eigenvalue problems

Here we distinguish as a special case those problems in which both $A$ and $B$ are symmetric and $B$ is positive-definite and well-conditioned with respect to inversion (i.e., all the eigenvalues of $B$ are significantly greater than zero). Such problems can be satisfactorily treated by first reducing them to case (ii) of
Section 2.1.1 and then using the methods described there to compute the eigensolutions. If $B$ is factorized as $LL^T$ ($L$ lower triangular), then $Ax = \lambda Bx$ is equivalent to the standard symmetric problem $Ry = \lambda y$, where $R = L A(L^T)$ and $y = L^T x$. After finding an eigenvector $y$ of $R$, the required $x$ is computed by back-substitution in $y = L^T x$.

For generalized problems of the form $Ax = \lambda Bx$ which do not fall into the special case, the QZ algorithm is provided.

In order to appreciate the domain in which this algorithm is appropriate we remark first that when $B$ is non-singular the problem $Ax = \lambda Bx$ is fully equivalent to the problem $(B^{-1} A)x = \lambda x$; both the eigenvalues and eigenvectors being the same. When $A$ is non-singular $Ax = \lambda Bx$ is equivalent to the problem $(A^{-1} B)x = \mu x$; the eigenvalues $\mu$ being the reciprocals of the required eigenvalues and the eigenvectors remaining the same. In theory then, provided at least one of the matrices $A$ and $B$ is non-singular, the generalized problem $Ax = \lambda Bx$ could be solved via the standard problem $Cx = \lambda x$ with an appropriate matrix $C$, and as far as economy of effort is concerned this is quite satisfactory. However, in practice, for this reduction to be satisfactory from the standpoint of numerical stability, one requires more than the mere non-singularity of $A$ or $B$. It is necessary that $B^{-1} A$ (or $A^{-1} B$) should not only exist but that $B$ (or $A$) should be well-conditioned with respect to inversion. The nearer $B$ (or $A$) is to singularity the more unsatisfactory $B^{-1} A$ (or $A^{-1} B$) will be as a vehicle for determining the required eigenvalues. Unfortunately one cannot counter ill-conditioning in $B$ (or $A$) by computing $B^{-1} A$ (or $A^{-1} B$) accurately to single precision using iterative refinement. Well-determined eigenvalues of the original $Ax = \lambda Bx$ may be poorly determined even by the correctly rounded version of $B^{-1} A$ (or $A^{-1} B$). The situation may in some instances be saved by the observation that if $Ax = \lambda Bx$ then $(A-kB)x = (\lambda-k)Bx$. Hence if $A-kB$ is non-singular we may solve the standard problem $[(A-kB)^{-1} B]x = (\mu)_k x$ and for numerical stability we require only that $(A-kB)$ be well-conditioned with
respect to inversion.

In practice one may well be in a situation where no $k$ is known for which $(A-kB)$ is well-conditioned with respect to inversion and indeed $(A-kB)$ may be singular for all $k$. The QZ algorithm is designed to deal directly with the problem $Ax=(\lambda)Bx$ itself and its performance is unaffected by singularity or near-singularity of $A$, $B$ or $A-kB$.

2.1.3. Sparse symmetric problems

If the matrices $A$ and $B$ are large and sparse (i.e., only a small proportion of the elements are non-zero), then the methods described in the previous Section are unsuitable, because in reducing the problem to a simpler form, much of the sparsity of the problem would be lost; hence the computing time and the storage required would be very large. Instead, for symmetric problems, the method of simultaneous iteration may be used to determine selected eigenvalues and the corresponding eigenvectors. The routine provided has been designed to handle both symmetric and generalized symmetric problems.

2.2. Singular Value Problems

The singular value decomposition of an $m$ by $n$ real matrix $A$ is given by

$$A = QDP^T,$$

where $Q$ is an $m$ by $m$ orthogonal matrix, $P$ is an $n$ by $n$ orthogonal matrix and $D$ is an $m$ by $n$ diagonal matrix with non-negative diagonal elements. The first $k=\min(m,n)$ columns of $Q$ and $P$ are the left- and right-hand singular vectors of $A$ and the $k$ diagonal elements of $D$ are the singular values.

When $A$ is complex then the singular value decomposition is given by

$$A = QDP^H,$$

$$A = QDP^T$$

where $Q$ and $P$ are unitary, $P^H$ denotes the complex conjugate of $P$ and $D$ is as above for the real case.

If the matrix $A$ has column means of zero, then $AP$ is the matrix of principal components of $A$ and the singular values are the square roots of the sample variances of the observations with respect to the principal components. (See also Chapter G03.)
Routines are provided to return the singular values and vectors of a general real or complex matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Discussion

There is one routine, F02FJF, which is designed for sparse symmetric eigenvalue problems, either standard or generalized. The remainder of the routines are designed for dense matrices.

3.2. Eigenvalue and Eigenvector Routines

These reduce the matrix $A$ to a simpler form by a similarity transformation $S^{-1}AS=T$ where $T$ is an upper Hessenberg or tridiagonal matrix, compute the eigensolutions of $T$, and then recover the eigenvectors of $A$ via the matrix $S$. The eigenvectors are normalised so that

$$
\sum_{r=1}^{n} |x_r|^2 = 1
$$

where $x_r$ is the $r$th component of the eigenvector $x$, and so that the $r$th element of largest modulus is real if $x$ is complex. For problems of the type $Ax = \lambda Bx$ with $A$ and $B$ symmetric and $B$ positive-definite, the eigenvectors are normalised so that $x^T B x = 1$, $x$ always being real for such problems.

3.3. Singular Value and Singular Vector Routines

These reduce the matrix $A$ to real bidiagonal form, $B$ say, by orthogonal transformations $Q^T A P = B$ in the real case, and by unitary transformations $Q^H A P = B$ in the complex case, and the singular values and vectors are computed via this bidiagonal form. The singular values are returned in descending order.

3.4. Decision Trees

(i) Eigenvalues and Eigenvectors
Please see figure in printed Reference Manual

(ii) Singular Values and Singular Vectors

Please see figure in printed Reference Manual

F02 -- Eigenvalues and Eigenvectors

Chapter F02

Eigenvalues and Eigenvectors

F02AAF All eigenvalues of real symmetric matrix

F02ABF All eigenvalues and eigenvectors of real symmetric matrix

F02ADF All eigenvalues of generalized real symmetric-definite eigenproblem

F02AEF All eigenvalues and eigenvectors of generalized real symmetric-definite eigenproblem

F02AFF All eigenvalues of real matrix

F02AGF All eigenvalues and eigenvectors of real matrix

F02AJF All eigenvalues of complex matrix

F02AKF All eigenvalues and eigenvectors of complex matrix

F02AWF All eigenvalues of complex Hermitian matrix

F02AXF All eigenvalues and eigenvectors of complex Hermitian matrix

F02BBF Selected eigenvalues and eigenvectors of real symmetric matrix

F02BJF All eigenvalues and optionally eigenvectors of generalized eigenproblem by QZ algorithm, real matrices

F02FJF Selected eigenvalues and eigenvectors of sparse symmetric eigenproblem

F02WEF SVD of real matrix

F02XEF SVD of complex matrix
1. Purpose

F02AAF calculates all the eigenvalues of a real symmetric matrix.

2. Specification

SUBROUTINE F02AAF (A, IA, N, R, E, IFAIL)
INTEGER IA, N, IFAIL
DOUBLE PRECISION A(IA,N), R(N), E(N)

3. Description

This routine reduces the real symmetric matrix A to a real symmetric tridiagonal matrix using Householder’s method. The eigenvalues of the tridiagonal matrix are then determined using the QL algorithm.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. On exit: the elements of A below the diagonal are overwritten, and the rest of the array is unchanged.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F02AAF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: R(N) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues in ascending order.
5: E(N) -- DOUBLE PRECISION array  Workspace

6: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   Failure in F02AVF(*) indicating that more than 30*N
   iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the eigenvalues depends on the sensitivity of the
matrix to rounding errors produced in tridiagonalisation. For a
detailed error analysis see Wilkinson and Reinsch [1] pp 222 and
235.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the real symmetric matrix:

\[
\begin{pmatrix}
0.5 & 0.0 & 2.3 & -2.6 \\
0.0 & 0.5 & -1.4 & -0.7 \\
2.3 & -1.4 & 0.5 & 0.0 \\
-2.6 & -0.7 & 0.0 & 0.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors
F02ABF
F02ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02ABF calculates all the eigenvalues and eigenvectors of a real symmetric matrix.

2. Specification

SUBROUTINE F02ABF (A, IA, N, R, V, IV, E, IFAIL)
INTEGER IA, N, IV, IFAIL
DOUBLE PRECISION A(IA,N), R(N), V(IV,N), E(N)

3. Description

This routine reduces the real symmetric matrix A to a real symmetric tridiagonal matrix by Householder’s method. The eigenvalues and eigenvectors are calculated using the QL algorithm.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input
On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. See also Section 8.

2: IA -- INTEGER Input
On entry: the first dimension of the array A as declared in the (sub)program from which F02ABF is called.
Constraint: IA >= N.

3: N -- INTEGER Input
On entry: n, the order of the matrix A.

4: R(N) -- DOUBLE PRECISION array Output
On exit: the eigenvalues in ascending order.

5: V(IV,N) -- DOUBLE PRECISION array Output
On exit: the normalised eigenvectors, stored by columns; the ith column corresponds to the ith eigenvalue. The eigenvectors are normalised so that the sum of squares of
the elements is equal to 1.

6: IV -- INTEGER Input
On entry:
the first dimension of the array V as declared in the
(sub)program from which F02ABF is called.
Constraint: IV >= N.

7: E(N) -- DOUBLE PRECISION array Workspace

8: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
Failure in F02AMF(*) indicating that more than 30*N
iterations are required to isolate all the eigenvalues.

7. Accuracy

The eigenvectors are always accurately orthogonal but the
accuracy of the individual eigenvectors is dependent on their
inherent sensitivity to changes in the original matrix. For a
detailed error analysis see Wilkinson and Reinsch [1] pp 222 and
235.

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users’ Note for your
implementation, the routine may be called with the same actual
array supplied for parameters A and V, in which case the
eigenvectors will overwrite the original matrix. However this is
not standard Fortran 77, and may not work on all systems.

9. Example

To calculate all the eigenvalues and eigenvectors of the real
symmetric matrix:
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02 -- Eigenvalue and Eigenvectors
F02ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02ADF calculates all the eigenvalues of $A\mathbf{x}=(\lambda)B\mathbf{x}$, where $A$ is a real symmetric matrix and $B$ is a real symmetric positive-definite matrix.

2. Specification

```fortran
SUBROUTINE F02ADF (A, IA, B, IB, N, R, DE, IFAIL)
INTEGER IA, IB, N, IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), R(N), DE(N)
```

3. Description

The problem is reduced to the standard symmetric eigenproblem using Cholesky's method to decompose $B$ into triangular matrices, $B=LL^T$, where $L$ is lower triangular. Then $A\mathbf{x}=(\lambda)B\mathbf{x}$ implies $-(L^{-1}AL^{-T})(L^{-T}\mathbf{x})=(\lambda)(L^{-T}\mathbf{x})$; hence the eigenvalues of $A\mathbf{x}=(\lambda)B\mathbf{x}$ are those of $P\mathbf{y}=(\lambda)y$ where $P$ is the symmetric matrix $L^{-1}AL^{-T}$. Householder's method is used to tridiagonalise the matrix $P$ and the eigenvalues are then found using the QL algorithm.

4. References

5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array  Input/Output
   On entry: the upper triangle of the n by n symmetric matrix
   A. The elements of the array below the diagonal need not be
   set. On exit: the lower triangle of the array is
   overwritten. The rest of the array is unchanged.

2: IA -- INTEGER  Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which F02ADF is called.
   Constraint: IA >= N.

3: B(IB,N) -- DOUBLE PRECISION array  Input/Output
   On entry: the upper triangle of the n by n symmetric
   positive-definite matrix B. The elements of the array below
   the diagonal need not be set. On exit: the elements below
   the diagonal are overwritten. The rest of the array is
   unchanged.

4: IB -- INTEGER  Input
   On entry:
   the first dimension of the array B as declared in the
   (sub)program from which F02ADF is called.
   Constraint: IB >= N.

5: N -- INTEGER  Input
   On entry: n, the order of the matrices A and B.

6: R(N) -- DOUBLE PRECISION array  Output
   On exit: the eigenvalues in ascending order.

7: DE(N) -- DOUBLE PRECISION array  Workspace

8: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   Failure in F01AEF(*); the matrix B is not positive-definite
   possibly due to rounding errors.
IFAIL= 2
Failure in F02AVF(*), more than 30*N iterations are required
to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-
conditioned with respect to inversion, the eigenvalues could be
inaccurately determined. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the general symmetric
eigenproblem Ax=(\lambda) Bx where A is the symmetric matrix:

\[
\begin{pmatrix}
0.5 & 1.5 & 6.6 & 4.8 \\
1.5 & 6.5 & 16.2 & 8.6 \\
6.6 & 16.2 & 37.6 & 9.8 \\
4.8 & 8.6 & 9.8 & -17.1
\end{pmatrix}
\]

and B is the symmetric positive-definite matrix:

\[
\begin{pmatrix}
1 & 3 & 4 & 1 \\
3 & 13 & 16 & 11 \\
4 & 16 & 24 & 18 \\
1 & 11 & 18 & 27
\end{pmatrix}
\]

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
F02AEF calculates all the eigenvalues and eigenvectors of \( Ax = (\lambda) Bx \), where \( A \) is a real symmetric matrix and \( B \) is a real symmetric positive-definite matrix.

2. Specification

```plaintext
SUBROUTINE F02AEF (A, IA, B, IB, N, R, V, IV, DL, E, IFAIL)
INTEGER IA, IB, N, IV, IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), R(N), V(IV,N), DL(N), E
1 (N)
```

3. Description

The problem is reduced to the standard symmetric eigenproblem using Cholesky's method to decompose \( B \) into triangular matrices \( T \)

\[
B = \begin{pmatrix} L & 0 \\ 0 & L \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & A \end{pmatrix} \begin{pmatrix} L^{-T} & 0 \\ 0 & L^{-T} \end{pmatrix},
\]

where \( L \) is lower triangular. Then \( Ax = (\lambda) Bx \) implies

\[
-L^{-T} A L^{-T} x = (\lambda) L^{-T} x;
\]

hence the eigenvalues of \( Ax = (\lambda) Bx \) are those of \( Py = (\lambda) y \), where \( P \) is the symmetric matrix \( L^{-T} A L^{-T} \). Householder's method is used to tridiagonalise the matrix \( P \) and the eigenvalues are found using the QL algorithm. An eigenvector \( z \) of the derived problem is related to

\[
T
\]
an eigenvector \( x \) of the original problem by \( z = L^{-T} x \). The eigenvectors \( z \) are determined using the QL algorithm and are

\[
T
\]
normalised so that \( z z = 1 \); the eigenvectors of the original problem are then determined by solving \( L^{-T} x = z \), and are normalised

\[
T
\]
so that \( x B x = 1 \).

4. References


5. Parameters

1: \( A(IA,N) \) -- DOUBLE PRECISION array Input/Output

On entry: the upper triangle of the \( n \) by \( n \) symmetric matrix \( A \). The elements of the array below the diagonal need not be set. On exit: the lower triangle of the array is overwritten. The rest of the array is unchanged. See also Section 8.

2: \( IA \) -- INTEGER Input
On entry: the first dimension of the array A as declared in the 
(sub)program from which F02AEF is called. 
Constraint: IA >= N.

3: B(IB,N) -- DOUBLE PRECISION array Input/Output 
On entry: the upper triangle of the n by n symmetric 
positive-definite matrix B. The elements of the array below 
the diagonal need not be set. On exit: the elements below 
the diagonal are overwritten. The rest of the array is 
unchanged.

4: IB -- INTEGER Input 
On entry: 
the first dimension of the array B as declared in the 
(sub)program from which F02AEF is called. 
Constraint: IB >= N.

5: N -- INTEGER Input 
On entry: n, the order of the matrices A and B.

6: R(N) -- DOUBLE PRECISION array Output 
On exit: the eigenvalues in ascending order.

7: V(IV,N) -- DOUBLE PRECISION array Output 
On exit: the normalised eigenvectors, stored by columns; 
the ith column corresponds to the ith eigenvalue. The 
T 
eigenvectors x are normalised so that x Bx=1. See also 
Section 8.

8: IV -- INTEGER Input 
On entry: 
the first dimension of the array V as declared in the 
(sub)program from which F02AEF is called. 
Constraint: IV >= N.

9: DL(N) -- DOUBLE PRECISION array Workspace 

10: E(N) -- DOUBLE PRECISION array Workspace 

11: IFAIL -- INTEGER Input/Output 
On entry: IFAIL must be set to 0, -1 or 1. For users not 
familiar with this parameter (described in the Essential 
Introduction) the recommended value is 0. 
On exit: IFAIL = 0 unless the routine detects an error (see 
Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
   Failure in F01AEF(*); the matrix B is not positive-definite, possibly due to rounding errors.

IFAIL= 2
   Failure in F02AMF(*); more than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-conditioned with respect to inversion, the eigenvectors could be inaccurately determined. For a detailed error analysis see Wilkinson and Reinsch [1] pp 310, 222 and 235.

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users’ Note for your implementation, the routine may be called with the same actual array supplied for parameters A and V, in which case the eigenvectors will overwrite the original matrix A. However this is not standard Fortran 77, and may not work on all systems.

9. Example

To calculate all the eigenvalues and eigenvectors of the general symmetric eigenproblem Ax=(\lambda) Bx where A is the symmetric matrix:

\begin{align*}
(0.5 & 1.5 & 6.6 & 4.8) \\
(1.5 & 6.5 & 16.2 & 8.6) \\
(6.6 & 16.2 & 37.6 & 9.8) \\
(4.8 & 8.6 & 9.8 & -17.1)
\end{align*}

and B is the symmetric positive-definite matrix:

\begin{align*}
(1 & 3 & 4 & 1) \\
(3 & 13 & 16 & 11) \\
(4 & 16 & 24 & 18) \\
(1 & 11 & 18 & 27)
\end{align*}

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F02 -- Eigenvalue and Eigenvectors
F02AFF
F02AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AFF calculates all the eigenvalues of a real unsymmetric matrix.

2. Specification

SUBROUTINE F02AFF (A, IA, N, RR, RI, INTGER, IFAIL)
INTEGER IA, N, INTGER(N), IFAIL
DOUBLE PRECISION A(IA,N), RR(N), RI(N)

3. Description

The matrix A is first balanced and then reduced to upper Hessenberg form using stabilised elementary similarity transformations. The eigenvalues are then found using the QR algorithm for real Hessenberg matrices.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the n by n matrix A. On exit: the array is overwritten.

2: IA -- INTEGER Input
   On entry: the dimension of the array A as declared in the (sub)program
   from which F02AFF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: RR(N) -- DOUBLE PRECISION array Output
   On exit: the real parts of the eigenvalues.
5: \textbf{RI(N)} -- DOUBLE PRECISION array \hspace{1cm} \textbf{Output}
   On exit: the imaginary parts of the eigenvalues.

6: \textbf{INTEGER(N)} -- INTEGER array \hspace{1cm} \textbf{Output}
   On exit: INTEGER(i) contains the number of iterations used
   to find the \(i\)th eigenvalue. If INTEGER(i) is negative, the \(i\)
th eigenvalue is the second of a pair found simultaneously.

   Note that the eigenvalues are found in reverse order,
   starting with the \(n\)th.

7: \textbf{IFAIL} -- INTEGER \hspace{1cm} \textbf{Input/Output}
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   More than 30*N iterations are required to isolate all the
   eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and
the multiplicity of the roots. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to \(n\)

9. Example

To calculate all the eigenvalues of the real matrix:

\[
\begin{pmatrix}
1.5 & 0.1 & 4.5 & -1.5 \\
-22.5 & 3.5 & 12.5 & -2.5 \\
-2.5 & 0.3 & 4.5 & -2.5 \\
-2.5 & 0.1 & 4.5 & 2.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

---

F02 -- Eigenvalue and Eigenvectors  
F02AGF  
F02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AGF calculates all the eigenvalues and eigenvectors of a real unsymmetric matrix.

2. Specification

```plaintext
SUBROUTINE F02AGF (A, IA, N, RR, RI, VR, IVR, VI, IVI,  
                        INTEGER, IFAIL)
INTEGER IA, N, IVR, IVI, INTEGER(N), IFAIL
DOUBLE PRECISION A(IA,N), RR(N), RI(N), VR(IVR,N), VI
                        (IVI,N)
```

3. Description

The matrix $A$ is first balanced and then reduced to upper Hessenberg form using real stabilised elementary similarity transformations. The eigenvalues and eigenvectors of the Hessenberg matrix are calculated using the QR algorithm. The eigenvectors of the Hessenberg matrix are back-transformed to give the eigenvectors of the original matrix $A$.

4. References


5. Parameters

1: $A(IA,N)$ -- DOUBLE PRECISION array  
   Input/Output  
   On entry: the $n$ by $n$ matrix $A$. On exit: the array is overwritten.

2: $IA$ -- INTEGER  
   Input  
   On entry:  
   the first dimension of the array $A$ as declared in the (sub)program from which F02AGF is called.  
   Constraint: $IA >= N$.  

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: RR(N) -- DOUBLE PRECISION array Output
   On exit: the real parts of the eigenvalues.

5: RI(N) -- DOUBLE PRECISION array Output
   On exit: the imaginary parts of the eigenvalues.

6: VR(IVR,N) -- DOUBLE PRECISION array Output
   On exit: the real parts of the eigenvectors, stored by
   columns. The ith column corresponds to the ith eigenvalue.
   The eigenvectors are normalised so that the sum of the
   squares of the moduli of the elements is equal to 1 and the
   element of largest modulus is real. This ensures that real
   eigenvalues have real eigenvectors.

7: IVR -- INTEGER Input
   On entry:
   the first dimension of the array VR as declared in the
   (sub)program from which F02AGF is called.
   Constraint: IVR >= N.

8: VI(IVI,N) -- DOUBLE PRECISION array Output
   On exit: the imaginary parts of the eigenvectors, stored by
   columns. The ith column corresponds to the ith eigenvalue.

9: IVI -- INTEGER Input
   On entry:
   the first dimension of the array VI as declared in the
   (sub)program from which F02AGF is called.
   Constraint: IVI >= N.

10: INTGER(N) -- INTEGER array Output
    On exit: INTGER(i) contains the number of iterations used
    to find the ith eigenvalue. If INTGER(i) is negative, the i
    th eigenvalue is the second of a pair found simultaneously.

    Note that the eigenvalues are found in reverse order,
    starting with the nth.

11: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the real matrix:

\[
\begin{pmatrix}
1.5 & 0.1 & 4.5 & -1.5 \\
-22.5 & 3.5 & 12.5 & -2.5 \\
-2.5 & 0.3 & 4.5 & -2.5 \\
-2.5 & 0.1 & 4.5 & 2.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02 -- Eigenvalue and Eigenvectors F02AJF
F02AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AJF calculates all the eigenvalues of a complex matrix.

2. Specification

SUBROUTINE F02AJF (AR, IAR, AI, IAI, N, RR, RI, INTGER,
1 INTEGER IAR, IAI, N, INTGER(N), IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N)

3. Description

The complex matrix A is first balanced and then reduced to upper Hessenberg form using stabilised elementary similarity transformations. The eigenvalues are then found using the modified LR algorithm for complex Hessenberg matrices.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input/Output
On entry: the real parts of the elements of the n by n complex matrix A. On exit: the array is overwritten.

2: IAR -- INTEGER Input
On entry: the first dimension of the array AR as declared in the (sub)program from which F02AJF is called.
Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input/Output
On entry: the imaginary parts of the elements of the n by n complex matrix A. On exit: the array is overwritten.

4: IAI -- INTEGER Input
On entry: the first dimension of the array AI as declared in the (sub)program from which F02AJF is called.
Constraint: IAI >= N.

5: N -- INTEGER Input
On entry: n, the order of the matrix A.

6: RR(N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvalues.

7: RI(N) -- DOUBLE PRECISION array Output
On exit: the imaginary parts of the eigenvalues.

8: INTGER(N) -- INTEGER array Workspace

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 401.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the complex matrix:

\((-21.0-5.0i, 24.60i, 13.6+10.2i, 4.0i)\)
\((22.5i, 26.00-5.00i, 7.5-10.0i, 2.5)\)
\((-2.0+1.5i, 1.68+2.24i, 4.5-5.0i, 1.5+2.0i)\).
\((-2.5i, -2.60, -2.7+3.6i, 2.5-5.0i)\)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F02AKF calculates all the eigenvalues and eigenvectors of a complex matrix.

2. Specification

```fortran
SUBROUTINE F02AKF (AR, IAR, AI, IAI, N, RR, RI, VR, IVR,
1       VI, IVI, INTEGER, IFAIL)
  INTEGER IAR, IAI, N, IVR, IVI, INTEGER(N), IFAIL
  DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N), VR
1       (IVR,N), VI(IVI,N)
```

3. Description

The complex matrix A is first balanced and then reduced to upper Hessenberg form by stabilised elementary similarity transformations. The eigenvalues and eigenvectors of the Hessenberg matrix are calculated using the LR algorithm. The eigenvectors of the Hessenberg matrix are back-transformed to give the eigenvectors of the original matrix.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real parts of the elements of the n by n complex matrix A. On exit: the array is overwritten.

2: IAR -- INTEGER Input
   On entry: the first dimension of the array AR as declared in the (sub)program from which F02AKF is called.
   Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input/Output
   On entry: the imaginary parts of the elements of the n by n complex matrix A. On exit: the array is overwritten.

4: IAI -- INTEGER Input
   On entry: the first dimension of the array AI as declared in the (sub)program from which F02AKF is called.
   Constraint: IAI >= N.

5: N -- INTEGER Input
   On entry: n, the order of the matrix A.
6: RR(N) -- DOUBLE PRECISION array
   On exit: the real parts of the eigenvalues.

7: RI(N) -- DOUBLE PRECISION array
   On exit: the imaginary parts of the eigenvalues.

8: VR(IVR,N) -- DOUBLE PRECISION array
   On exit: the real parts of the eigenvectors, stored by
          columns. The ith column corresponds to the ith eigenvalue.
          The eigenvectors are normalised so that the sum of squares
          of the moduli of the elements is equal to 1 and the element
          of largest modulus is real.

9: IVR -- INTEGER
   On entry:
   the first dimension of the array VR as declared in the
   (sub)program from which F02AKF is called.
   Constraint: IVR >= N.

10: VI(IVI,N) -- DOUBLE PRECISION array
   On exit: the imaginary parts of the eigenvectors, stored by
           columns. The ith column corresponds to the ith eigenvalue.

11: IVI -- INTEGER
   On entry:
   the first dimension of the array VI as declared in the
   (sub)program from which F02AKF is called.
   Constraint: IVI >= N.

12: INTGER(N) -- INTEGER array
    Workspace

13: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not
           familiar with this parameter (described in the Essential
           Introduction) the recommended value is 0.
           On exit: IFAIL = 0 unless the routine detects an error (see
                    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   More than 30*N iterations are required to isolate all the
eigenvalues.

7. Accuracy
The accuracy of the results depends on the conditioning of the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the complex matrix:

\[
\begin{pmatrix}
-21.0-5.0i & 24.60i & 13.6+10.2i & 4.0i \\
22.5i & 26.00-5.00i & 7.5-10.0i & 2.5 \\
-2.0+1.5i & 1.68+2.24i & 4.5-5.0i & 1.5+2.0i \\
-2.5i & -2.60 & -2.7+3.6i & 2.5-5.0i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F02 -- Eigenvalue and Eigenvectors

F02AWF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AWF calculates all the eigenvalues of a complex Hermitian matrix.

2. Specification

```fortran
SUBROUTINE F02AWF (AR, IAR, AI, IAI, N, R, WK1, WK2, WK3, 
                   IFAIL)
INTEGER IAR, IAI, N, IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), WK1(N), 
                   WK2(N), WK3(N)
```

3. Description

The complex Hermitian matrix A is first reduced to a real tridiagonal matrix by n-2 unitary transformations, and a
subsequent diagonal transformation. The eigenvalues are then derived using the QL algorithm, an adaptation of the QR algorithm.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. On exit: the array is overwritten.

2: IAR -- INTEGER Input
   On entry: the first dimension of the array AR as declared in the (sub)program from which F02AWF is called.
   Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input/Output
   On entry: the imaginary parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. On exit: the array is overwritten.

4: IAI -- INTEGER Input
   On entry: the first dimension of the array AI as declared in the (sub)program from which F02AWF is called.
   Constraint: IAI >= N.

5: N -- INTEGER Input
   On entry: n, the order of the complex Hermitian matrix, A.

6: R(N) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues in ascending order.

7: WK1(N) -- DOUBLE PRECISION array Workspace

8: WK2(N) -- DOUBLE PRECISION array Workspace

9: WK3(N) -- DOUBLE PRECISION array Workspace
10: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL = 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

8. Further Comments
The time taken by the routine is approximately proportional to n.

9. Example
To calculate all the eigenvalues of the complex Hermitian matrix:

\[
\begin{pmatrix}
0.50 & 0.00 & 1.84+1.38i & 2.08-1.56i \\
0.00 & 0.50 & 1.12+0.84i & -0.56+0.42i \\
1.84-1.38i & 1.12-0.84i & 0.50 & 0.00 \\
2.08+1.56i & -0.56-0.42i & 0.00 & 0.50 \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02 -- Eigenvalue and Eigenvectors
F02AXF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
F02AXF calculates all the eigenvalues and eigenvectors of a complex Hermitian matrix.

2. Specification

SUBROUTINE F02AXF (AR, IAR, AI, IAI, N, R, VR, IVR, VI,
1           IVI, WK1, WK2, WK3, IFAIL)
INTEGER IAR, IAI, N, IVR, IVI, IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), VR(IVR,N), VI
1 (IVI,N), WK1(N), WK2(N), WK3(N)

3. Description

The complex Hermitian matrix is first reduced to a real tridiagonal matrix by n-2 unitary transformations and a subsequent diagonal transformation. The eigenvalues and eigenvectors are then derived using the QL algorithm, an adaptation of the QR algorithm.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array
   Input
   On entry: the real parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

2: IAR -- INTEGER
   Input
   On entry: the first dimension of the array AR as declared in the (sub)program from which F02AXF is called.
   Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array
   Input
   On entry: the imaginary parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

4: IAI -- INTEGER
   Input
   On entry:
the first dimension of the array AI as declared in the
(sub)program from which F02AXF is called.
Constraint: IAI >= N.

5: N -- INTEGER Input
On entry: n, the order of the matrix, A.

6: R(N) -- DOUBLE PRECISION array Output
On exit: the eigenvalues in ascending order.

7: VR(IVR,N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvectors, stored by
columns. The ith column corresponds to the ith eigenvector.
The eigenvectors are normalised so that the sum of the
squares of the moduli of the elements is equal to 1 and the
element of largest modulus is real. See also Section 8.

8: IVR -- INTEGER Input
On entry:
the first dimension of the array VR as declared in the
(sub)program from which F02AXF is called.
Constraint: IVR >= N.

9: VI(IVI,N) -- DOUBLE PRECISION array Output
On exit: the imaginary parts of the eigenvectors, stored by
columns. The ith column corresponds to the ith eigenvector.
See also Section 8.

10: IVI -- INTEGER Input
On entry:
the first dimension of the array VI as declared in the
(sub)program from which F02AXF is called.
Constraint: IVI >= N.

11: WK1(N) -- DOUBLE PRECISION array Workspace
12: WK2(N) -- DOUBLE PRECISION array Workspace
13: WK3(N) -- DOUBLE PRECISION array Workspace

14: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL = 1
More than 30*N iterations are required to isolate all the eigenvalues.

IFAIL = 2
The diagonal elements of AI are not all zero, i.e., the complex matrix is not Hermitian.

7. Accuracy
The eigenvectors are always accurately orthogonal but the accuracy of the individual eigenvalues and eigenvectors is dependent on their inherent sensitivity to small changes in the original matrix. For a detailed error analysis see Peters [1] page 3 and [2] page 3.

8. Further Comments
The time taken by the routine is approximately proportional to n
Unless otherwise stated in the implementation document, the routine may be called with the same actual array supplied for parameters AR and VR, and for AI and VI, in which case the eigenvectors will overwrite the original matrix A. However this is not standard Fortran 77, and may not work on all systems.

9. Example
To calculate the eigenvalues and eigenvectors of the complex Hermitian matrix:

\[
\begin{pmatrix}
0.50 & 0.00 & 1.84+1.38i & 2.08-1.56i \\
0.00 & 0.50 & 1.12+0.84i & -0.56+0.42i \\
1.84-1.38i & 1.12-0.84i & 0.50 & 0.00 \\
2.08+1.56i & -0.56-0.42i & 0.00 & 0.50 \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02BBF calculates selected eigenvalues and eigenvectors of a real symmetric matrix by reduction to tridiagonal form, bisection and inverse iteration, where the selected eigenvalues lie within a given interval.

2. Specification

```plaintext
INTEGER IA, N, M, MM, IV, ICOUNT(M), IFAIL
DOUBLE PRECISION A(IA,N), ALB, UB, R(M), V(IV,M), D(N), E(N), E2(N), X(N,7), G(N)
LOGICAL C(N)
```

3. Description

The real symmetric matrix A is reduced to a symmetric tridiagonal matrix T by Householder's method. The eigenvalues which lie within a given interval [l,u], are calculated by the method of bisection. The corresponding eigenvectors of T are calculated by inverse iteration. A back-transformation is then performed to obtain the eigenvectors of the original matrix A.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array
   Input/Output
   On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. On exit: the elements of A below the diagonal are overwritten, and the rest of the array is unchanged.

2: IA -- INTEGER
   Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F02BBF is called.
   Constraint: IA >= N.

3: N -- INTEGER
   Input
   On entry: n, the order of the matrix A.
4: ALB -- DOUBLE PRECISION
   Input
5: UB -- DOUBLE PRECISION
   Input
   On entry: l and u, the lower and upper end-points of the
   interval within which eigenvalues are to be calculated.
6: M -- INTEGER
   Input
   On entry: an upper bound for the number of eigenvalues
   within the interval.
7: MM -- INTEGER
   Output
   On exit: the actual number of eigenvalues within the
   interval.
8: R(M) -- DOUBLE PRECISION array
   Output
   On exit: the eigenvalues, not necessarily in ascending
   order.
9: V(IV,M) -- DOUBLE PRECISION array
   Output
   On exit: the eigenvectors, stored by columns. The ith
   column corresponds to the ith eigenvalue. The eigenvectors
   are normalised so that the sum of the squares of the
   elements are equal to 1.
10: IV -- INTEGER
    Input
    On entry:
    the first dimension of the array V as declared in the
    (sub)program from which F02BBF is called.
    Constraint: IV >= N.
11: D(N) -- DOUBLE PRECISION array
    Workspace
12: E(N) -- DOUBLE PRECISION array
    Workspace
13: E2(N) -- DOUBLE PRECISION array
    Workspace
14: X(N,7) -- DOUBLE PRECISION array
    Workspace
15: G(N) -- DOUBLE PRECISION array
    Workspace
16: C(N) -- LOGICAL array
    Workspace
17: ICOUNT(M) -- INTEGER array
    Output
    On exit: ICOUNT(i) contains the number of iterations for
    the ith eigenvalue.
18: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
M is less than the number of eigenvalues in the given interval. On exit MM contains the number of eigenvalues in the interval. Rerun with this value for M.

IFAIL = 2
More than 5 iterations are required to determine any one eigenvector.

7. Accuracy

There is no guarantee of the accuracy of the eigenvectors as the results depend on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 222 and 436.

8. Further Comments

The time taken by the routine is approximately proportional to n^3

This subroutine should only be used when less than 25% of the eigenvalues and the corresponding eigenvectors are required. Also this subroutine is less efficient with matrices which have multiple eigenvalues.

9. Example

To calculate the eigenvalues lying between -2.0 and 3.0, and the corresponding eigenvectors of the real symmetric matrix:

\[
\begin{pmatrix}
0.5 & 0.0 & 2.3 & -2.6 \\
0.0 & 0.5 & -1.4 & -0.7 \\
2.3 & -1.4 & 0.5 & 0.0 \\
-2.6 & -0.7 & 0.0 & 0.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02BJF calculates all the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem \( Ax = \lambda Bx \) where \( A \) and \( B \) are real, square matrices, using the QZ algorithm.

2. Specification

```fortran
SUBROUTINE F02BJF (N, A, IA, B, IB, EPS1, ALFR, ALFI,
1       BETA, MATV, V, IV, ITER, IFAIL)
INTEGER N, IA, IB, IV, ITER(N), IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), EPS1, ALFR(N), ALFI(N),
1       BETA(N), V(IV,N)
LOGICAL MATV
```

3. Description

All the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem \( Ax = \lambda Bx \) where \( A \) and \( B \) are real, square matrices, are determined using the QZ algorithm. The QZ algorithm consists of 4 stages:

(a) \( A \) is reduced to upper Hessenberg form and at the same time \( B \) is reduced to upper triangular form.

(b) \( A \) is further reduced to quasi-triangular form while the triangular form of \( B \) is maintained.

(c) The quasi-triangular form of \( A \) is reduced to triangular form and the eigenvalues extracted. This routine does not actually produce the eigenvalues \( \lambda \), but instead returns \( \alpha_j \) and \( \beta_j \) such that

\[
\lambda_j = \frac{\alpha_j}{\beta_j}, \quad j = 1, 2, \ldots, n
\]

The division by \( \beta_j \) becomes the responsibility of the user's program, since \( \beta_j \) may be zero indicating an infinite eigenvalue. Pairs of complex eigenvalues occur
with \((\alpha_j)/(\beta_j)\) and \((\alpha_{j+1})/(\beta_{j+1})\) complex
j j j+1 j+1
conjugates, even though \((\alpha_j)\) and \((\alpha_{j+1})\) are not
j j+1
conjugate.

(d) If the eigenvectors are required \((\text{MATV} = \text{.TRUE.})\), they are
obtained from the triangular matrices and then transformed
back into the original co-ordinate system.

4. References

241--256.


Algorithm. Linear Algebra and Appl. 28 285--303.

5. Parameters

1: \(N\) -- INTEGER Input
   On entry: \(n\), the order of the matrices \(A\) and \(B\).

2: \(A(IA,N)\) -- DOUBLE PRECISION array Input/Output
   On entry: the \(n\) by \(n\) matrix \(A\). On exit: the array is
   overwritten.

3: \(IA\) -- INTEGER Input
   On entry:
   the first dimension of the array \(A\) as declared in the
   (sub)program from which F02BJF is called.
   Constraint: \(IA \geq N\).

4: \(B(IB,N)\) -- DOUBLE PRECISION array Input/Output
   On entry: the \(n\) by \(n\) matrix \(B\). On exit: the array is
   overwritten.

5: \(IB\) -- INTEGER Input
   On entry:
   the first dimension of the array \(B\) as declared in the
   (sub)program from which F02BJF is called.
   Constraint: \(IB \geq N\).

6: \(EPS1\) -- DOUBLE PRECISION Input
   On entry: the tolerance used to determine negligible
elements. If \( \text{EPS1} > 0.0 \), an element will be considered negligible if it is less than \( \text{EPS1} \) times the norm of its matrix. If \( \text{EPS1} \leq 0.0 \), machine precision is used in place of \( \text{EPS1} \). A positive value of \( \text{EPS1} \) may result in faster execution but less accurate results.

7: \text{ALFR(N)} -- DOUBLE PRECISION array
Output

8: \text{ALFI(N)} -- DOUBLE PRECISION array
Output
On exit: the real and imaginary parts of (alpha), for
\( j = 1, 2, \ldots, n \).

9: \text{BETA(N)} -- DOUBLE PRECISION array
Output
On exit: (beta), for \( j = 1, 2, \ldots, n \).

10: MATV -- LOGICAL
Input
On entry: MATV must be set .TRUE. if the eigenvectors are required, otherwise .FALSE. .

11: \text{V(IV,N)} -- DOUBLE PRECISION array
Output
On exit: if MATV = .TRUE., then
(i) if the jth eigenvalue is real, the jth column of V contains its eigenvector;
(ii) if the jth and (j+1)th eigenvalues form a complex pair, the jth and (j+1)th columns of V contain the real and imaginary parts of the eigenvector associated with the first eigenvalue of the pair. The conjugate of this vector is the eigenvector for the conjugate eigenvalue.

Each eigenvector is normalised so that the component of largest modulus is real and the sum of squares of the moduli equal one.

If MATV = .FALSE., V is not used.

12: IV -- INTEGER
Input
On entry: the first dimension of the array V as declared in the (sub)program from which F02BJF is called.
Constraint: \( IV \geq N \).

13: \text{ITER(N)} -- INTEGER array
Output
On exit: \text{ITER}(j) contains the number of iterations needed to obtain the jth eigenvalue. Note that the eigenvalues are obtained in reverse order, starting with the nth.

14: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= i

More than 30*N iterations are required to determine all the diagonal 1 by 1 or 2 by 2 blocks of the quasi-triangular form in the second step of the QZ algorithm. IFAIL is set to the index i of the eigenvalue at which this failure occurs. If the soft failure option is used, (alpha) and (beta) are j j correct for j=i+1,i+2,...,n, but V does not contain any correct eigenvectors.

7. Accuracy

The computed eigenvalues are always exact for a problem (A+E)x=(lambda)(B+F)x where ||E||/||A|| and ||F||/||B|| are both of the order of max(EPS1,(epsilon)), EPS1 being defined as in Section 5 and (epsilon) being the machine precision.

Note: interpretation of results obtained with the QZ algorithm often requires a clear understanding of the effects of small changes in the original data. These effects are reviewed in Wilkinson [3], in relation to the significance of small values of (alpha) and (beta). It should be noted that if (alpha) and j j (beta) are both small for any j, it may be that no reliance can j be placed on any of the computed eigenvalues (lambda)=(alpha)/(beta). The user is recommended to study [3] i i and, if in difficulty, to seek expert advice on determining the sensitivity of the eigenvalues to perturbations in the data.

8. Further Comments

The time taken by the routine is approximately proportional to n and also depends on the value chosen for parameter EPS1.

9. Example
To find all the eigenvalues and eigenvectors of $Ax = \lambda Bx$ where

$$
\begin{pmatrix}
3.9 & 12.5 & -34.5 & -0.5 \\
4.3 & 21.5 & -47.5 & 7.5 \\
4.3 & 21.5 & -43.5 & 3.5 \\
4.4 & 26.0 & -46.0 & 6.0
\end{pmatrix}
$$

and

$$
\begin{pmatrix}
1 & 2 & -3 & 1 \\
1 & 3 & -5 & 4 \\
1 & 3 & -4 & 3 \\
1 & 3 & -4 & 4
\end{pmatrix}
$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

F02 -- Eigenvalue and Eigenvectors

F02FJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

To find eigenvalues and eigenvectors of a real sparse symmetric or generalized symmetric eigenvalue problem.

2. Specification

```fortran
SUBROUTINE F02FJF (N, M, K, NOITS, TOL, DOT, IMAGE, MONIT,
1  NOVECS, X, NRX, D, WORK, LWORK, RWORK,
2  LRWORK, IWORK, LIWORK, IFAIL)
INTEGER N, M, K, NOITS, NOVECS, NRX, LWORK, LIWORK, IFAIL
DOUBLE PRECISION TOL, DOT, X(NRX,K), D(K), WORK(LWORK),
1  RWORK(LRWORK)
EXTERNAL DOT, IMAGE, MONIT
```

3. Description

F02FJF finds the $m$ eigenvalues of largest absolute value and the corresponding eigenvectors for the real eigenvalue problem.
\[ Cx = (\lambda)x \] (1)

where \( C \) is an \( n \) by \( n \) matrix such that

\[ \quad T \\
\quad BC = C B \] (2)

for a given positive-definite matrix \( B \). \( C \) is said to be \( B \)-symmetric. Different specifications of \( C \) allow for the solution of a variety of eigenvalue problems. For example, when

\[ \quad T \\
\quad C = A \quad \text{and} \quad B = I \quad \text{where} \quad A = A \]

the routine finds the \( m \) eigenvalues of largest absolute magnitude for the standard symmetric eigenvalue problem

\[ Ax = (\lambda)x. \] (3)

The routine is intended for the case where \( A \) is sparse.

As a second example, when

\[ \quad -1 \\
\quad C = B^{-1} A \]

where

\[ \quad T \\
\quad A = A \]

the routine finds the \( m \) eigenvalues of largest absolute magnitude for the generalized symmetric eigenvalue problem

\[ Ax = (\lambda)Bx. \] (4)

The routine is intended for the case where \( A \) and \( B \) are sparse.

The routine does not require \( C \) explicitly, but \( C \) is specified via a user-supplied routine \( \text{IMAGE} \) which, given an \( n \) element vector \( z \), computes the image \( w \) given by

\[ w = Cz. \]

\[ -1 \]

For instance, in the above example, where \( C = B^{-1} A \), routine \( \text{IMAGE} \) will need to solve the positive-definite system of equations \( Bw = Az \) for \( w \).

To find the \( m \) eigenvalues of smallest absolute magnitude of (3)
we can choose $C=A^{-1}$ and hence find the reciprocals of the required eigenvalues, so that IMAGE will need to solve $Aw=z$ for $w$, and correspondingly for (4) we can choose $C=A^{-1}B$ and solve $Aw=Bz$ for $w$.

A table of examples of choice of IMAGE is given in Table 3.1. It should be remembered that the routine also returns the corresponding eigenvectors and that $B$ is positive-definite. Throughout $A$ is assumed to be symmetric and, where necessary, non-singularity is also assumed.

<table>
<thead>
<tr>
<th>Eigenvalues Required</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Ax=(\lambda)x$</td>
<td>$(B=I)Ax=(\lambda)x$ $ABx=(\lambda)x$</td>
</tr>
<tr>
<td>Largest Compute</td>
<td>Solve</td>
</tr>
<tr>
<td>$w=Az$</td>
<td>$Bw=Az$</td>
</tr>
<tr>
<td>Smallest Solve</td>
<td>Solve</td>
</tr>
<tr>
<td>(Find $Aw=z$)</td>
<td>$A\nu=Az$</td>
</tr>
<tr>
<td>Furthest Compute</td>
<td>Solve</td>
</tr>
<tr>
<td>(Find $w=(A-(\sigma)I)z$)</td>
<td>$Bw=(A-(\sigma)B)z$</td>
</tr>
<tr>
<td>Closest to Solve</td>
<td>Solve</td>
</tr>
<tr>
<td>(Find $1/((\lambda)-(\sigma))w=z$)</td>
<td>$(A-(\sigma)B)w=Bz$</td>
</tr>
</tbody>
</table>

Table 3.1
The Requirement of IMAGE for Various Problems

The matrix $B$ also need not be supplied explicitly, but is specified via a user-supplied routine DOT which, given $n$ element $T$ vectors $z$ and $w$, computes the generalized dot product $w^T Bz$.

F02FJF is based upon routine SIMITZ (see Nikolai [1]), which is itself a derivative of the Algol procedure ritzit (see Rutishauser [4]), and uses the method of simultaneous (subspace)
iteration. (See Parlett [2] for description, analysis and advice on the use of the method.)

The routine performs simultaneous iteration on \( k > m \) vectors. Initial estimates to \( p \leq k \) eigenvectors, corresponding to the \( p \) eigenvalues of \( C \) of largest absolute value, may be supplied by the user to F02FJF. When possible \( k \) should be chosen so that the \( k \)th eigenvalue is not too close to the \( m \) required eigenvalues, but if \( k \) is initially chosen too small then F02FJF may be re-entered, supplying approximations to the \( k \) eigenvectors found so far and with \( k \) then increased.

At each major iteration F02FJF solves an \( r \) by \( r \) (\( r \leq k \)) eigenvalue sub-problem in order to obtain an approximation to the eigenvalues for which convergence has not yet occurred. This approximation is refined by Chebyshev acceleration.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
   On entry: \( n \), the order of the matrix \( C \). Constraint: \( N \geq 1 \).

2: \( M \) -- INTEGER Input/Output
   On entry: \( m \), the number of eigenvalues required.
   Constraint: \( M \geq 1 \). On exit: \( m \), the number of eigenvalues actually found. It is equal to \( m \) if IFAIL = 0 on exit, and is less than \( m \) if IFAIL = 2, 3 or 4. See Section 6 and Section 8 for further information.

3: \( K \) -- INTEGER Input
   On entry: the number of simultaneous iteration vectors to be used. Too small a value of \( K \) may inhibit convergence, while a larger value of \( K \) incurs additional storage and additional work per iteration. Suggested value: \( K = M + 4 \) will often be
a reasonable choice in the absence of better information. 
Constraint: M < K <= N.

4: NOITS -- INTEGER            Input/Output 
On entry: the maximum number of major iterations (eigenvalue 
sub-problems) to be performed. If NOITS <= 0, then the value 
100 is used in place of NOITS. On exit: the number of 
iterations actually performed.

5: TOL -- DOUBLE PRECISION          Input 
On entry: a relative tolerance to be used in accepting 
eigenvalues and eigenvectors. If the eigenvalues are 
required to about t significant figures, then TOL should be 
-\[10^{-t}\] 
set to about 10^-i . d is accepted as an eigenvalue as soon 
\[\frac{|d_i| \times TOL}{10}\] 
as two successive approximations to d differ by less than 
\[\frac{(d_f - d_i)}{d_i}\] < TOL, where f is the 
normalised residual of the current approximation to the 
eigenvector (see Section 8 for further information). The 
values of the f and d can be printed from routine MONIT.

If TOL is supplied outside the range ((\epsilon), 1.0), where 
(\epsilon) is the machine precision, then the value (\epsilon) 
is used in place of TOL.

6: DOT -- DOUBLE PRECISION FUNCTION, supplied by the user.        
External Procedure 
DOT must return the value w \cdot z for given vectors w and z. 
For the standard eigenvalue problem, where B=I, DOT must 
return the dot product w \cdot z.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION DOT (IFLAG, N, Z, W,} \\
1\text{  RWORK, LRWORK,} \\
2\text{  IWORK, LIWORK)} \\
\text{INTEGER IFLAG, N, LRWORK, IWORK(LIWORK),} \\
1\text{  LIWORK} \\
\text{DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)}
\]

1: IFLAG -- INTEGER            Input/Output
On entry: IFLAG is always non-negative. On exit: IFLAG
may be used as a flag to indicate a failure in the
\( T \)
computation of \( w Bz \). If IFLAG is negative on exit from
DOT, then F02FJF will exit immediately with IFAIL set
to IFLAG. Note that in this case DOT must still be
assigned a value.

2: \( N \) -- INTEGER
   Input
   On entry: the number of elements in the vectors \( z \) and \( w \)
   and the order of the matrix \( B \).

3: \( Z(N) \) -- DOUBLE PRECISION array
   Input
   On entry: the vector \( z \) for which \( w Bz \) is required.

4: \( W(N) \) -- DOUBLE PRECISION array
   Input
   On entry: the vector \( w \) for which \( w Bz \) is required.

5: \( RWORK(LRWORK) \) -- DOUBLE PRECISION array
   User Workspace

6: \( LRWORK \) -- INTEGER
   Input

7: \( IWORK(LIWORK) \) -- INTEGER array
   User Workspace

8: \( LIWORK \) -- INTEGER
   Input
   DOT is called from F02FJF with the parameters RWORK,
   LRWORK, IWORK and LIWORK as supplied to F02FJF. The
   user is free to use the arrays RWORK and IWORK to
   supply information to DOT and to IMAGE as an
   alternative to using COMMON.
   DOT must be declared as EXTERNAL in the (sub)program
   from which F02FJF is called. Parameters denoted as
   Input must not be changed by this procedure.

7: \( IMAGE \) -- SUBROUTINE, supplied by the user.
   External Procedure
   IMAGE must return the vector \( w=Cz \) for a given vector \( z \).

Its specification is:

```fortran
SUBROUTINE IMAGE (IFLAG, N, Z, W, RWORK, LRWORK,
                   1   IWORK, LIWORK)
    INTEGER    IFLAG, N, LRWORK, IWORK(LIWORK),
    1   LIWORK
    DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)
```
1: IFLAG -- INTEGER
   Input/Output
   On entry: IFLAG is always non-negative. On exit: IFLAG
   may be used as a flag to indicate a failure in the
   computation of v. If IFLAG is negative on exit from
   IMAGE, then F02FJF will exit immediately with IFAIL set
to IFLAG.

2: N -- INTEGER
   Input
   On entry: n, the number of elements in the vectors w
   and z, and the order of the matrix C.

3: Z(N) -- DOUBLE PRECISION array
   Input
   On entry: the vector z for which Cz is required.

4: W(N) -- DOUBLE PRECISION array
   Output
   On exit: the vector w=Cz.

5: RWORK(LRWORK) -- DOUBLE PRECISION array
   User Workspace

6: LRWORK -- INTEGER
   Input

7: IWORK(LIWORK) -- INTEGER array
   User Workspace

8: LIWORK -- INTEGER
   Input
   IMAGE is called from F02FJF with the parameters RWORK,
   LRWORK, IWORK and LIWORK as supplied to F02FJF. The
   user is free to use the arrays RWORK and IWORK to
   supply information to IMAGE and DOT as an alternative
   to using COMMON.
   IMAGE must be declared as EXTERNAL in the (sub)program
   from which F02FJF is called. Parameters denoted as
   Input must not be changed by this procedure.

8: MONIT -- SUBROUTINE, supplied by the user.
   External Procedure
   MONIT is used to monitor the progress of F02FJF. MONIT may
   be the dummy subroutine F02FJZ if no monitoring is actually
   required. (F02FJZ is included in the NAG Foundation Library
   and so need not be supplied by the user. The routine name
   F02FJZ may be implementation dependent: see the Users’ Note
   for your implementation for details.) MONIT is called after
   the solution of each eigenvalue sub-problem and also just
   prior to return from F02FJF. The parameters ISTATE and
   NEXTIT allow selective printing by MONIT.

   Its specification is:

   SUBROUTINE MONIT (ISTATE, NEXTIT, NEVALS,
   1       NEVECS, K, F, D)
INTEGER ISTATE, NEXTIT, NEVALS, NEVECS, K
DOUBLE PRECISION F(K), D(K)

1: ISTATE -- INTEGER Input
On entry: ISTATE specifies the state of F02FJF and will have values as follows:
ISTATE = 0
   No eigenvalue or eigenvector has just been accepted.
ISTATE = 1
   One or more eigenvalues have been accepted since the last call to MONIT.
ISTATE = 2
   One or more eigenvectors have been accepted since the last call to MONIT.
ISTATE = 3
   One or more eigenvalues and eigenvectors have been accepted since the last call to MONIT.
ISTATE = 4
   Return from F02FJF is about to occur.

2: NEXTIT -- INTEGER Input
On entry: the number of the next iteration.

3: NEVALS -- INTEGER Input
On entry: the number of eigenvalues accepted so far.

4: NEVECS -- INTEGER Input
On entry: the number of eigenvectors accepted so far.

5: K -- INTEGER Input
On entry: k, the number of simultaneous iteration vectors.

6: F(K) -- DOUBLE PRECISION array Input
On entry: a vector of error quantities measuring the state of convergence of the simultaneous iteration vectors. See the parameter TOL of F02FJF above and Section 8 for further details. Each element of F is initially set to the value 4.0 and an element remains at 4.0 until the corresponding vector is tested.

7: D(K) -- DOUBLE PRECISION array Input
On entry: D(i) contains the latest approximation to the absolute value of the ith eigenvalue of C.
MONIT must be declared as EXTERNAL in the (sub)program from which F02FJF is called. Parameters denoted as Input must not be changed by this procedure.

9: NOVECS -- INTEGER
   On entry: the number of approximate vectors that are being supplied in X. If NOVECS is outside the range (0,K), then the value 0 is used in place of NOVECS.

10: X(NRX,K) -- DOUBLE PRECISION array
    On entry: if 0 < NOVECS <= K, the first NOVECS columns of X must contain approximations to the eigenvectors corresponding to the NOVECS eigenvalues of largest absolute value of C. Supplying approximate eigenvectors can be useful when reasonable approximations are known, or when the routine is being restarted with a larger value of K. Otherwise it is not necessary to supply approximate vectors, as simultaneous iteration vectors will be generated randomly by the routine. On exit: if IFAIL = 0, 2, 3 or 4, the first m' columns contain the eigenvectors corresponding to the eigenvalues returned in the first m' elements of D (see below); and the next k-m'-1 columns contain approximations to the eigenvectors corresponding to the approximate eigenvalues returned in the next k-m'-1 elements of D. Here m' is the value returned in M (see above), the number of eigenvalues actually found. The kth column is used as workspace.

11: NRX -- INTEGER
    On entry: the first dimension of the array X as declared in the (sub)program from which F02FJF is called. Constraint: NRX >= N.

12: D(K) -- DOUBLE PRECISION array
    On exit: if IFAIL = 0, 2, 3 or 4, the first m' elements contain the first m' eigenvalues in decreasing order of magnitude; and the next k-m'-1 elements contain approximations to the next k-m'-1 eigenvalues. Here m' is the value returned in M (see above), the number of eigenvalues actually found. D(k) contains the value e where (-e,e) is the latest interval over which Chebyshev acceleration is performed.

13: WORK(LWORK) -- DOUBLE PRECISION array
    Workspace

14: LWORK -- INTEGER
    On entry: the length of the array WORK, as declared in the (sub)program from which F02FJF is called. Constraint: LWORK>=3*K+max(K*K,2*N).
15: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace
RWORK is not used by F02FJF, but is passed directly to
routines DOT and IMAGE and may be used to supply information
to these routines.

16: LRWORK -- INTEGER Input
On entry: the length of the array RWORK, as declared in the
(sub)program from which F02FJF is called. Constraint: LRWORK
\geq 1.

17: IWORK(LIWORK) -- INTEGER array User Workspace
IWORK is not used by F02FJF, but is passed directly to
routines DOT and IMAGE and may be used to supply information
to these routines.

18: LIWORK -- INTEGER Input
On entry: the length of the array IWORK, as declared in the
(sub)program from which F02FJF is called. Constraint: LIWORK
\geq 1.

19: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL ≠ 0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
the output of an error message when soft failure occurs, set
IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL< 0
A negative value of IFAIL indicates an exit from F02FJF
because the user has set IFLAG negative in DOT or IMAGE. The
value of IFAIL will be the same as the user's setting of
IFLAG.

IFAIL= 1
On entry N < 1,
or \( M < 1 \),
or \( M \geq K \),
or \( K > N \),
or \( NRX < N \),
or \( LWORK < 3K + \max(K\times K\times N) \),
or \( LRWORK < 1 \),
or \( LIWORK < 1 \).

\textbf{IFAIL} = 2

Not all the requested eigenvalues and vectors have been obtained. Approximations to the \( r \)th eigenvalue are oscillating rapidly indicating that severe cancellation is occurring in the \( r \)th eigenvector and so \( M \) is returned as \((r-1)\). A restart with a larger value of \( K \) may permit convergence.

\textbf{IFAIL} = 3

Not all the requested eigenvalues and vectors have been obtained. The rate of convergence of the remaining eigenvectors suggests that more than \( \text{NOITS} \) iterations would be required and so the input value of \( M \) has been reduced. A restart with a larger value of \( K \) may permit convergence.

\textbf{IFAIL} = 4

Not all the requested eigenvalues and vectors have been obtained. \( \text{NOITS} \) iterations have been performed. A restart, possibly with a larger value of \( K \), may permit convergence.

\textbf{IFAIL} = 5

This error is very unlikely to occur, but indicates that convergence of the eigenvalue sub-problem has not taken place. Restarting with a different set of approximate vectors may allow convergence. If this error occurs the user should check carefully that \texttt{F02FJF} is being called correctly.

7. Accuracy

Eigenvalues and eigenvectors will normally be computed to the accuracy requested by the parameter \( \text{TOL} \), but eigenvectors corresponding to small or to close eigenvalues may not always be computed to the accuracy requested by the parameter \( \text{TOL} \). Use of the routine \texttt{MONIT} to monitor acceptance of eigenvalues and eigenvectors is recommended.
8. Further Comments

The time taken by the routine will be principally determined by the time taken to solve the eigenvalue sub-problem and the time taken by the routines DOT and IMAGE. The time taken to solve an eigenvalue sub-problem is approximately proportional to \( nk^2 \). It is important to be aware that several calls to DOT and IMAGE may occur on each major iteration.

As can be seen from Table 3.1, many applications of F02FJF will require routine IMAGE to solve a system of linear equations. For example, to find the smallest eigenvalues of \( Ax = \lambda Bx \), IMAGE needs to solve equations of the form \( Aw = Bz \) for \( w \) and routines from Chapters F01 and F04 of the NAG Foundation Library will frequently be useful in this context. In particular, if \( A \) is a positive-definite variable band matrix, F04MCF may be used after \( A \) has been factorized by F01MCF. Thus factorization need be performed only once prior to calling F02FJF. An illustration of this type of use is given in the example program in Section 9.

An approximation \( \hat{d}_i \), to the \( i \)th eigenvalue, is accepted as soon as \( \hat{d}_i \) and the previous approximation differ by less than \( \hat{h} \)

\[
|\hat{d}_i| \times TOL/10.
\]

Eigenvectors are accepted in groups corresponding to clusters of eigenvalues that are equal, or nearly equal, in absolute value and that have already been accepted. If \( \hat{d}_i \) is the last eigenvalue in such a group and we define the residual \( r_j \) as

\[
r_j =Cx_j - y_j
\]

where \( y_j \) is the projection of \( Cx_j \), with respect to \( B \), onto the space spanned by \( x_1, x_2, \ldots, x_r \) and \( x_r \) is the current approximation to the \( j \)th eigenvector, then the value \( f \) returned in MONIT is given by

\[
f = \max \left( \frac{|r_j^T r_j|}{||C|| ||x_j||} \right) = x_j^T B x_j
\]
and each vector in the group is accepted as an eigenvector if

\[
\frac{|d|f}{|d|-e} < TOL
\]

where \(e\) is the current approximation to \(|d|\). The values of the \(k\) 
\(f\) are systematically increased if the convergence criteria 
i appear to be too strict. See Rutishauser [4] for further details.

The algorithm implemented by F02FJF differs slightly from SIMITZ 
(Nikolai [1]) in that the eigenvalue sub-problem is solved using 
the singular value decomposition of the upper triangular matrix \(R^T\) 
of the Gram-Schmidt factorization of \(Cx\), rather than forming \(RR^T\) 
r

9. Example

To find the four eigenvalues of smallest absolute value and 
corresponding eigenvectors for the generalized symmetric 
eigenvalue problem \(Ax = \lambda Bx\), where \(A\) and \(B\) are the 16 by 16 
matrices

\[
A = \begin{pmatrix}
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
1 & a & a & a \\
\end{pmatrix}
\]

where \(a=-4\) 

\[
(1 \ b)
\]
TOL is taken as 0.0001 and 6 iteration vectors are used. F01MAF is used to factorize the matrix A, prior to calling F02FJF, and F04MAF is used within IMAGE to solve the equations Aw=Bz for w. Details of the factorization of A are passed from F01MAF to F04MAF by means of the COMMON block BLOCK1.

Output from MONIT occurs each time ISTATE is non-zero. Note that the required eigenvalues are the reciprocals of the eigenvalues returned by F02FJF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F02 -- Eigenvalue and Eigenvectors  
F02WEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02WEF returns all, or part, of the singular value decomposition of a general real matrix.

2. Specification
3. Description

The \( m \) by \( n \) matrix \( A \) is factorized as

\[
T \\
A = QDP,
\]

where

\[
D = \begin{cases} 
    (S) & \text{for } m > n, \\
    D = (0), & \text{for } m = n, \\
    D = (S\ 0), & \text{for } m < n,
\end{cases}
\]

\( Q \) is an \( m \) by \( m \) orthogonal matrix, \( P \) is an \( n \) by \( n \) orthogonal matrix and \( S \) is a \( \min(m,n) \) by \( \min(m,n) \) diagonal matrix with non-negative diagonal elements, \( sv_1, sv_2, \ldots, sv_{\min(m,n)} \), ordered such that

\[
sv_1 \geq sv_2 \geq \ldots \geq sv_{\min(m,n)} \geq 0.
\]

The first \( \min(m,n) \) columns of \( Q \) are the left-hand singular vectors of \( A \), the diagonal elements of \( S \) are the singular values of \( A \) and the first \( \min(m,n) \) columns of \( P \) are the right-hand singular vectors of \( A \).

Either or both of the left-hand and right-hand singular vectors of \( A \) may be requested and the matrix \( C \) given by

\[
T \\
C = Q B,
\]

where \( B \) is an \( m \) by \( ncolb \) given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing \( A \) to upper triangular form by means of Householder transformations, from the left when \( m \geq n \) and from the right when \( m < n \). The upper triangular form is then reduced to bidiagonal form by Givens plane rotations and finally the QR algorithm is used to
obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] DSVDC.

Note that if $K$ is any orthogonal diagonal matrix so that

$$
T
$$

$$
KK = I,
$$

(so that $K$ has elements $+1$ or $-1$ on the diagonal)

then

$$
T
$$

$$
A = (QK)D(PK)
$$

is also a singular value decomposition of $A$.

4. References


5. Parameters

1: $M$ -- INTEGER
   Input
   On entry: the number of rows, $m$, of the matrix $A$.
   Constraint: $M >= 0$.
   When $M = 0$ then an immediate return is effected.

2: $N$ -- INTEGER
   Input
   On entry: the number of columns, $n$, of the matrix $A$.
   Constraint: $N >= 0$.
   When $N = 0$ then an immediate return is effected.

3: $A(LDA,*)$ -- DOUBLE PRECISION array
   Input/Output
   Note: the second dimension of the array $A$ must be at least $\text{max}(1,N)$.
   On entry: the leading $m$ by $n$ part of the array $A$ must
contain the matrix A whose singular value decomposition is required. On exit: if M >= N and WANTQ = .TRUE., then the leading m by n part of A will contain the first n columns of the orthogonal matrix Q.

If M < N and WANTP = .TRUE., then the leading m by n part of A will contain the first m rows of the orthogonal matrix P.

If M >= N and WANTQ = .FALSE. and WANTP = .TRUE., then the leading n by n part of A will contain the first n rows of the orthogonal matrix P.

Otherwise the leading m by n part of A is used as internal workspace.

4: LDA -- INTEGER Input
On entry: the first dimension of the array A as declared in the (sub)program from which F02WEF is called.
Constraint: LDA >= max(1,M).

5: NCOLB -- INTEGER Input
On entry: ncolb, the number of columns of the matrix B. When NCOLB = 0 the array B is not referenced. Constraint: NCOLB >= 0.

6: B(LDB,*) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array B must be at least max(1,ncolb) On entry: if NCOLB > 0, the leading m by ncolb part of the array B must contain the matrix to be transformed. On exit: B is overwritten by the m by ncolb T matrix Q B.

7: LDB -- INTEGER Input
On entry: the first dimension of the array B as declared in the (sub)program from which F02WEF is called.
Constraint: if NCOLB > 0 then LDB >= max(1,M).

8: WANTQ -- LOGICAL Input
On entry: WANTQ must be .TRUE., if the left-hand singular vectors are required. If WANTQ = .FALSE., then the array Q is not referenced.

9: Q(LDQ,*) -- DOUBLE PRECISION array Output
Note: the second dimension of the array Q must be at least max(1,M).
On exit: if $M < N$ and WANTQ = .TRUE., the leading $m$ by $m$ part of the array Q will contain the orthogonal matrix Q. Otherwise the array Q is not referenced.

10: LDQ -- INTEGER
Input
On entry:
the first dimension of the array Q as declared in the (sub)program from which F02WEF is called.
Constraint: if $M < N$ and WANTQ = .TRUE., LDQ >= max(1,M).

11: SV(*) -- DOUBLE PRECISION array
Output
Note: the length of SV must be at least min(M,N). On exit:
the min(M,N) diagonal elements of the matrix S.

12: WANTP -- LOGICAL
Input
On entry: WANTP must be .TRUE. if the right-hand singular vectors are required. If WANTP = .FALSE., then the array PT is not referenced.

13: PT(LDPT,*) -- DOUBLE PRECISION array
Output
Note: the second dimension of the array PT must be at least max(1,N).
On exit: if $M >= N$ and WANTQ and WANTP are .TRUE., the leading $n$ by $n$ part of the array PT will contain the $T$ orthogonal matrix P. Otherwise the array PT is not referenced.

14: LDPT -- INTEGER
Input
On entry:
the first dimension of the array PT as declared in the (sub)program from which F02WEF is called.
Constraint: if $M >= N$ and WANTQ and WANTP are .TRUE., LDPT >= max(1,N).

15: WORK(*) -- DOUBLE PRECISION array
Output
Note: the length of WORK must be at least max(1,lwork), where lwork must be as given in the following table:

$M >= N$
WANTQ is .TRUE. and WANTP = .TRUE.

2
lwork=max(N +5*(N-1),N+NCOLB,4)

WANTQ = .TRUE. and WANTP = .FALSE.

2
lwork=max(N +4*(N-1),N+NCOLB,4)

WANTQ = .FALSE. and WANTP = .TRUE.
lwork=max(3*(N-1),2) when NCOLB = 0
\text{lwork} = \max(5(N-1), 2) \text{ when } NCOLB > 0

\text{WANTQ} = \text{.FALSE.} \text{ and } \text{WANTP} = \text{.FALSE.}
\text{lwork} = \max(2(N-1), 2) \text{ when } NCOLB = 0
\text{lwork} = \max(3(N-1), 2) \text{ when } NCOLB > 0

M < N
\text{WANTQ} = \text{.TRUE.} \text{ and } \text{WANTP} = \text{.TRUE.}
\text{lwork} = \max(M + 5(M-1), 2)
\text{WANTQ} = \text{.TRUE.} \text{ and } \text{WANTP} = \text{.FALSE.}
\text{lwork} = \max(3(M-1), 1)
\text{WANTQ} = \text{.FALSE.} \text{ and } \text{WANTP} = \text{.TRUE.}
\text{lwork} = \max(M + 3(M-1), 2) \text{ when } NCOLB = 0
\text{lwork} = \max(M + 5(M-1), 2) \text{ when } NCOLB > 0

\text{WANTQ} = \text{.FALSE.} \text{ and } \text{WANTP} = \text{.FALSE.}
\text{lwork} = \max(2(M-1), 1) \text{ when } NCOLB = 0
\text{lwork} = \max(3(M-1), 1) \text{ when } NCOLB > 0

On exit: \text{WORK(min(M, N))} contains the total number of
iterations taken by the R algorithm.

The rest of the array is used as workspace.

16: \text{IFAIL} -- INTEGER
\text{Input/Output}
On entry: \text{IFAIL} must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: \text{IFAIL} = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \text{IFAIL} = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

\text{IFAIL}=-1
One or more of the following conditions holds:
M < 0,
N < 0,
LDA < M,
NCOLB < 0,
LDB < M and NCOLB > 0,
LDQ < M and M < N and WANTQ = .TRUE.,
LDPT < N and M >= N and WANTQ = .TRUE., and WANTP = .TRUE..

IFAIL > 0
The QR algorithm has failed to converge in 50*min(m,n) iterations. In this case SV(1), SV(2),..., SV(IFAIL) may not have been found correctly and the remaining singular values may not be the smallest. The matrix A will nevertheless have been factorized as A = QEP, where the leading min(m,n) by min(m,n) part of E is a bidiagonal matrix with SV(1), SV(2),..., SV(min(m,n)) as the diagonal elements and WORK(1), WORK(2),..., WORK(min(m,n)-1) as the super-diagonal elements.

This failure is not likely to occur.

7. Accuracy
The computed factors Q, D and P satisfy the relation

T
QDP = A + E,

where

||E|| <= c(epsilon)||A||,

(epsilon) being the machine precision, c is a modest function of m and n and ||.|| denotes the spectral (two) norm. Note that ||A|| = sv.

8. Further Comments
Following the use of this routine the rank of A may be estimated by a call to the INTEGER FUNCTION F06KLF(*). The statement:

IRANK = F06KLF(MIN(M, N), SV, 1, TOL)
returns the value (k-1) in IRANK, where k is the smallest integer for which SV(k)<tol*SV(1), where tol is the tolerance supplied in TOL, so that IRANK is an estimate of the rank of S and thus also of A. If TOL is supplied as negative then the machine precision is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

\[
\begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}
\]

T

together with the vector Q b for the vector

\[
\begin{pmatrix}
1.1 \\
0.9 \\
0.6 \\
0.0 \\
-0.8
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

9.2. Example 2

To find the singular value decomposition of the 3 by 5 matrix

\[
\begin{pmatrix}
2.0 & 2.0 & 1.6 & 2.0 & 1.2 \\
2.5 & 2.5 & -0.4 & -0.5 & -0.3 \\
2.5 & 2.5 & 2.8 & 0.5 & -2.9
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors
F02XEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02XEF returns all, or part, of the singular value decomposition of a general complex matrix.

2. Specification

```fortran
SUBROUTINE F02XEF (M, N, A, LDA, NCOLB, B, LDB, WANTQ, Q, 1 LDQ, SV, WANTP, PH, LDPH, RWORK, CWORK, 2 IFAIL)

INTEGER M, N, LDA, NCOLB, LDB, LDQ, LDPH, 1 IFAIL
DOUBLE PRECISION SV(*), RWORK(*)
COMPLEX(KIND=KIND(1.0D0)) A(LDA,*), B(LDB,*), Q(LDQ,*), 1 PH(LDPH,*), CWORK(*)
LOGICAL WANTQ, WANTP
```

3. Description

The m by n matrix A is factorized as

\[
H \quad A = QDP ,
\]

where

\[
\begin{align*}
\text{(S)} \\
D &= (0) & m & > n, \\
D &= S, & m &= n, \\
D &= (S \ 0), & m & < n,
\end{align*}
\]

Q is an m by m unitary matrix, P is an n by n unitary matrix and S is a min(m,n) by min(m,n) diagonal matrix with real non-negative diagonal elements, \( s_1, s_2, \ldots, s_{\text{min}(m,n)} \), ordered such that

\[
s_1 \geq s_2 \geq \ldots \geq s_{\text{min}(m,n)} \geq 0.
\]

The first min(m,n) columns of Q are the left-hand singular vectors of A, the diagonal elements of S are the singular values of A and the first min(m,n) columns of P are the right-hand singular vectors of A.
Either or both of the left-hand and right-hand singular vectors of $A$ may be requested and the matrix $C$ given by

$$
C = Q B,
$$

where $B$ is an $m$ by $n$ column given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing $A$ to upper triangular form by means of Householder transformations, from the left when $m \geq n$ and from the right when $m < n$. The upper triangular form is then reduced to bidiagonal form by Givens plane rotations and finally the QR algorithm is used to obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] ZSVDC.

Note that if $K$ is any unitary diagonal matrix so that

$$
K K = I,
$$

then

$$
A = (QK)D(PK)
$$

is also a singular value decomposition of $A$.

4. References


5. Parameters

1: M -- INTEGER Input

On entry: the number of rows, $m$, of the matrix $A$.

Constraint: $M \geq 0$.

When $M = 0$ then an immediate return is effected.
2: N -- INTEGER  
   Input
   On entry: the number of columns, n, of the matrix A.
   Constraint: N >= 0.

   When N = 0 then an immediate return is effected.

3: A(LDA,*), -- COMPLEX(KIND(1.0D)) array  
   Input/Output
   Note: the second dimension of the array A must be at least
   max(1,N).
   On entry: the leading m by n part of the array A must
   contain the matrix A whose singular value decomposition is
   required. On exit: if M >= N and WANTQ = .TRUE., then the
   leading m by n part of A will contain the first n columns of
   the unitary matrix Q.
   If M < N and WANTP = .TRUE., then the leading m by n part of
   H
   A will contain the first m rows of the unitary matrix P.
   will contain the first m rows of the unitary matrix P If M
   >= N and WANTQ = .FALSE. and WANTP = .TRUE., then the
   leading n by n part of A will contain the first n
   rows of the unitary matrix P. Otherwise the leading m by n
   part of A is used as internal workspace.

4: LDA -- INTEGER  
   Input
   On entry: the first dimension of the array A as declared in the
   (sub)program from which F02XEF is called.
   Constraint: LDA >= max(1,M).

5: NCOLB -- INTEGER  
   Input
   On entry: ncolb, the number of columns of the matrix B.
   When NCOLB = 0 the array B is not referenced. Constraint:
   NCOLB >= 0.

6: B(LDB,*), -- COMPLEX(KIND(1.0D)) array  
   Input/Output
   Note: the second dimension of the array B must be at least
   max(1,NCOLB).
   On entry: if NCOLB > 0, the leading m by ncolb part of the
   array B must contain the matrix to be transformed. On exit:
   H
   B is overwritten by the m by ncolb matrix Q B.

7: LDB -- INTEGER  
   Input
   On entry: the first dimension of the array B as declared in the
   (sub)program from which F02XEF is called.
   Constraint: if NCOLB > 0, then LDB >= max(1,M).
8: WANTQ -- LOGICAL  
   Input
   On entry: WANTQ must be .TRUE. if the left-hand singular
   vectors are required. If WANTQ = .FALSE. then the array Q is
   not referenced.

9: Q(LDQ,*) -- COMPLEX(KIND(1.0D)) array  
   Output
   Note: the second dimension of the array Q must be at least
   max(1,M).
   On exit: if M < N and WANTQ = .TRUE., the leading m by m
   part of the array Q will contain the unitary matrix Q.
   Otherwise the array Q is not referenced.

10: LDQ -- INTEGER  
    Input
    On entry:
    the first dimension of the array Q as declared in the
    (sub)program from which F02XEF is called.
    Constraint: if M < N and WANTQ = .TRUE., LDQ >= max(1,M).

11: SV(*) -- DOUBLE PRECISION array  
    Output
    Note: the length of SV must be at least min(M,N). On exit:
    the min(m,n) diagonal elements of the matrix S.

12: WANTP -- LOGICAL  
    Input
    On entry: WANTP must be .TRUE. if the right-hand singular
    vectors are required. If WANTP = .FALSE. then the array PH
    is not referenced.

13: PH(LDPH,*) -- DOUBLE PRECISION array  
    Output
    Note: the second dimension of the array PH must be at least
    max(1,N).
    On exit: if M >= N and WANTQ and WANTP are .TRUE., the
    leading n by n part of the array PH will contain the unitary
    matrix P . Otherwise the array PH is not referenced.

14: LDPH -- INTEGER  
    Input
    On entry:
    the first dimension of the array PH as declared in the
    (sub)program from which F02XEF is called.
    Constraint: if M >= N and WANTQ and WANTP are .TRUE., LDPH
    >= max(1,N).

15: RWORK(*) -- DOUBLE PRECISION array  
    Output
    Note: the length of RWORK must be at least max(1,lrwork),
    where lrwork must satisfy:
    lrwork=2*(min(M,N)-1) when
    NCOLB = 0 and WANTQ and WANTP are .TRUE.,
    lrwork=3*(min(M,N)-1) when
    either NCOLB = 0 and WANTQ = .FALSE. and WANTP = .

TRUE., or WANTP = .FALSE. and one or both of NCOLB > 0 and WANTQ = .TRUE.

lrwork=5*(min(M,N)-1)
otherwise.
On exit: RWORK(min(M,N)) contains the total number of iterations taken by the QR algorithm.

The rest of the array is used as workspace.

16: CWORK(*) -- COMPLEX(KIND(1.0D)) array Workspace
Note: the length of CWORK must be at least max(1,lcwork), where lcwork must satisfy:

2
lcwork=N+max(N ,NCOLB) when M >= N and WANTQ and WANTP are both .TRUE.

2
lcwork=N+max(N +N,NCOLB) when M >= N and WANTQ = .TRUE., but WANTP = .FALSE.

lcwork=N+max(N,NCOLB) when M >= N and WANTQ = .FALSE.

2
lcwork=M +M when M < N and WANTP = .TRUE.

lcwork = M when M < N and WANTP = .FALSE.

17: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1
One or more of the following conditions holds:
M < 0,
\( N < 0, \)
\( LDA < M, \)
\( NCOLB < 0, \)
\( LDB < M \) and \( NCOLB > 0, \)
\( LDQ < M \) and \( M < N \) and \( WANTQ = .TRUE. , \)
\( LDPH < N \) and \( M \geq N \) and \( WANTQ = .TRUE. \) and \( WANTP = .TRUE. . \)

\( IFAIL > 0 \)

The \( QR \) algorithm has failed to converge in \( 50 \times \min(m,n) \) iterations. In this case \( SV(1), SV(2), \ldots, SV(IFAIL) \) may not have been found correctly and the remaining singular values may not be the smallest. The matrix \( A \) will nevertheless have been factorized as \( A = QEP \) where the leading \( \min(m,n) \) by \( \min(m,n) \) part of \( E \) is a bidiagonal matrix with \( SV(1), SV(2), \ldots, SV(\min(m,n)) \) as the diagonal elements and \( RWORK(1), RWORK(2), \ldots, RWORK(\min(m,n)-1) \) as the super-diagonal elements.

This failure is not likely to occur.

7. Accuracy

The computed factors \( Q, D \) and \( P \) satisfy the relation

\[
H \quad QDP = A + E,
\]

where

\[
||E|| \leq c(\epsilon)||A||,
\]

(\( \epsilon \)) being the machine precision, \( c \) is a modest function of \( m \) and \( n \) and \( ||.|| \) denotes the spectral (two) norm. Note that \( ||A|| = sv \).

8. Further Comments

Following the use of this routine the rank of \( A \) may be estimated by a call to the INTEGER FUNCTION F06KLF(*). The statement:

\[
\text{IRANK} = \text{F06KLF}(\text{MIN}(M, N), SV, 1, \text{TOL})
\]
returns the value \((k-1)\) in IRANK, where \(k\) is the smallest integer for which \(SV(k) < tol \times SV(1)\), where \(tol\) is the tolerance supplied in TOL, so that IRANK is an estimate of the rank of \(S\) and thus also of \(A\). If TOL is supplied as negative then the machine precision is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

\[
\begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.0i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

\(A\)

H
together with the vector \(Qb\) for the vector

\[
\begin{pmatrix}
-0.55+1.05i \\
0.49+0.93i \\
0.56-0.16i \\
0.39+0.23i \\
1.13+0.83i
\end{pmatrix}
\]

\(b\)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

9.2. Example 2

To find the singular value decomposition of the 3 by 5 matrix

\[
\begin{pmatrix}
0.5i & 0.4-0.3i & 0.4 & 0.3+0.4i & 0.3i \\
-0.5-1.5i & 0.9-1.3i & -0.4-0.4i & 0.1-0.7i & 0.3-0.3i \\
-1.0-1.0i & 0.2-1.4i & 1.8 & 0.0 & -2.4i
\end{pmatrix}
\]

\(A\)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagEigenPackage (NAGF02)

Exports:
- f02aaf
- f02abf
- f02adf
- f02aef
- f02aff
- f02agf
- f02ajf
- f02akf
- f02awf
- f02axf
- f02bbf
- f02bjf
- f02fjf
- f02wef
- f02xef

--- package NAGF02 NagEigenPackage ---

)abbrev package NAGF02 NagEigenPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:20 1994
++ Description:
++ This package uses the NAG Library to compute
++ eigenvalues and eigenvectors of a matrix
++ eigenvalues and eigenvectors of generalized matrix
++ eigenvalue problems
++ singular values and singular vectors of a matrix.

NagEigenPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports == with
  f02aaf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
    ++ f02aaf(ia,n,a,ifail)
    ++ calculates all the eigenvalue.
    ++ See \downlink{Manual Page}{manpageXXf02aaf}.
  f02abf : (Matrix DoubleFloat,Integer,Integer,Integer,Integer) -> Result
    ++ f02abf(a,ia,n,iv,ifail)
    ++ calculates all the eigenvalues of a real
    ++ symmetric matrix.
    ++ See \downlink{Manual Page}{manpageXXf02abf}.
  f02adf : (Integer,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f02adf(ia,ib,n,a,b,ifail)
++ calculates all the eigenvalues of \( Ax=\lambda Bx \), where \( A \)
++ is a real symmetric matrix and \( B \) is a real symmetric positive-
++ definite matrix.
++ See \downlink{Manual Page}{manpageXXf02adf}.

f02aef : (Integer,Integer,Integer,Integer,Matrix
DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f02aef(ia,ib,n,iv,a,b,ifail)
++ calculates all the eigenvalues of
++ \( Ax=\lambda Bx \), where \( A \) is a real symmetric matrix and \( B \) is a
++ real symmetric positive-definite matrix.
++ See \downlink{Manual Page}{manpageXXf02aef}.

f02aff : (Integer,Integer,Integer,Integer,Matrix
DoubleFloat,Integer) -> Result
++ f02aff(ia,n,a,ifail)
++ calculates all the eigenvalues of a real unsymmetric
++ matrix.
++ See \downlink{Manual Page}{manpageXXf02aff}.

f02agf : (Integer,Integer,Integer,Integer,Matrix
DoubleFloat,Integer) -> Result
++ f02agf(ia,n,ivr,ivi,a,ifail)
++ calculates all the eigenvalues of a real
++ unsymmetric matrix.
++ See \downlink{Manual Page}{manpageXXf02agf}.

f02ajf : (Integer,Integer,Integer,Integer,Integer,
Matrix DoubleFloat,Integer) -> Result
++ f02ajf(iar,iai,n,ar,ai,ifail)
++ calculates all the eigenvalues.
++ See \downlink{Manual Page}{manpageXXf02ajf}.

f02akf : (Integer,Integer,Integer,Integer,Integer,Integer,
Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f02akf(iar,iai,n,ivr,ivi,ar,ai,ifail)
++ calculates all the eigenvalues of a complex matrix.
++ See \downlink{Manual Page}{manpageXXf02akf}.

f02awf : (Integer,Integer,Integer,Integer,Integer,
Matrix DoubleFloat,Integer) -> Result
++ f02awf(iar,iai,n,ar,ai,ifail)
++ calculates all the eigenvalues of a complex Hermitian
++ matrix.
++ See \downlink{Manual Page}{manpageXXf02awf}.

f02axf : (Matrix DoubleFloat,Integer,Integer,Integer,
Matrix DoubleFloat,Integer,Integer,Integer,Integer) -> Result
++ f02axf(ar,iar,ai,iai,n,ivr,ivi,ifail)
++ calculates all the eigenvalues of a complex Hermitian
++ matrix.
++ See \downlink{Manual Page}{manpageXXf02axf}.

f02bbf : (Integer,Integer,Integer,Integer,Integer,
Matrix DoubleFloat,Integer) -> Result
++ f02bbf(ia,n,alb,ub,m,iv,a,ifail)
++ calculates selected eigenvalues of a real
CHAPTER 15. CHAPTER N

++ symmetric matrix by reduction to tridiagonal form, bisection and
++ inverse iteration, where the selected eigenvalues lie within a
++ given interval.
++ See \downlink{Manual Page}{manpageXXf02bbf}.  

f02bjf : (Integer,Integer,Integer,DoubleFloat,
        Boolean,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f02bjf(n,ia,ib,eps1,matv,iv,a,b,ifail)
++ calculates all the eigenvalues and, if required, all the
++ eigenvectors of the generalized eigenproblem \( Ax = \lambda Bx \)
++ where A and B are real, square matrices, using the QZ algorithm.
++ See \downlink{Manual Page}{manpageXXf02bjf}.

f02fjf : (Integer,Integer,DoubleFloat,Integer,
        Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,
        Integer,Union(fn:FileName,fp:Asp27(DOT)),
        Union(fn:FileName,fp:Asp28(IMAGE))) -> Result
++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
        liwork,m,noits,x,ifail,dot,image)
++ finds eigenvalues of a real sparse symmetric
++ or generalized symmetric eigenvalue problem.
++ See \downlink{Manual Page}{manpageXXf02fjf}.

f02fjf : (Integer,Integer,DoubleFloat,Integer,
        Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,
        Integer,Union(fn:FileName,fp:Asp27(DOT)),
        Union(fn:FileName,fp:Asp28(IMAGE)),FileName) -> Result
++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
        liwork,m,noits,x,ifail,dot,image,monit)
++ finds eigenvalues of a real sparse symmetric
++ or generalized symmetric eigenvalue problem.
++ See \downlink{Manual Page}{manpageXXf02fjf}.

f02wef : (Integer,Integer,Integer,Integer,
        Integer,Boolean,Integer,Boolean,Integer,Matrix DoubleFloat,
        Matrix DoubleFloat,Integer) -> Result
++ f02wef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldpt,a,b,ifail)
++ returns all, or part, of the singular value decomposition
++ of a general real matrix.
++ See \downlink{Manual Page}{manpageXXf02wef}.

f02xef : (Integer,Integer,Integer,Integer,
        Integer,Boolean,Integer,Boolean,Integer,Matrix Complex DoubleFloat,
        Matrix Complex DoubleFloat,Integer) -> Result
++ f02xef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldph,a,b,ifail)
++ returns all, or part, of the singular value decomposition
++ of a general complex matrix.
++ See \downlink{Manual Page}{manpageXXf02xef}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(DoubleFloat)

f02aaf(iaArg::Integer,nArg::Integer,aArg::Matrix DoubleFloat,_
  ifailArg::Integer): Result ==
  [(invokeNagman(NIL$Lisp,
    "f02aaf",_
    ["r":S,"e":S]$Lisp,
    ["double":S,["r":S,"n":S]$Lisp,[_
      ["e":S,"n":S]$Lisp]$Lisp,
      ["integer":S,"ia":S,"ifail":S]$Lisp,_
    ]$Lisp),_
      ["r":S,"a":S,"ifail":S]$Lisp,_
      [(iaArg::Any,nArg::Any,ifailArg::Any,aArg::Any)$Lisp]_]
  pretend List (Record(key:Symbol,entry:Any))$Result

f02abf(aArg::Matrix DoubleFloat,iaArg::Integer,nArg::Integer,_
  ivArg::Integer,ifailArg::Integer): Result ==
  [(invokeNagman(NIL$Lisp,
    "f02abf",_
    ["r":S,"v":S,"e":S]$Lisp,
    ["double":S,["a":S,"ia":S,"n":S]$Lisp,_
      ["r":S,"n":S]$Lisp,[_
        ["v":S,"iv":S,"n":S]$Lisp,_
        ["e":S,"n":S]$Lisp]$Lisp,
      ]$Lisp,_
    ]$Lisp,_
    ["r":S,"v":S,"ifail":S]$Lisp,_
    [(iaArg::Any,nArg::Any,ivArg::Any,ifailArg::Any,aArg::Any)_,
    @List Any]$Lisp]$Lisp,
  pretend List (Record(key:Symbol,entry:Any))$Result

f02adf(iaArg::Integer,ibArg::Integer,nArg::Integer,_
  aArg::Matrix DoubleFloat,bArg::Matrix DoubleFloat,_
  ifailArg::Integer): Result ==
  [(invokeNagman(NIL$Lisp,
    "f02adf",_
    ["r":S,"de":S]$Lisp,_
  ]
    [(invokeNagman(NIL$Lisp,
        "f02aef",
        ["ia"::S,"ib"::S,"n"::S,"iv"::S,"ifail"::S,
         ["r"::S,"v"::S,"dl"::S,"e"::S]$Lisp,
         ["double"::S, ["r"::S, "n"::S]$Lisp, ["v"::S, "n"::S]$Lisp,
         ["a"::S, "ia"::S, "n"::S]$Lisp, ["b"::S, "ib"::S, "n"::S]$Lisp,
         ["dl"::S, "n"::S]$Lisp, ["e"::S, "n"::S]$Lisp]$Lisp,
        ["integer"::S, ["ia"::S, "ib"::S, "n"::S,$Lisp,
         "iv"::S, "ifail"::S]$Lisp,$Lisp)
    pretend List (Record(key:Symbol, entry:Any)))$Result

f02aff(iaArg: Integer, nArg: Integer, aArg: Matrix DoubleFloat, ifailArg: Integer): Result == 
    [(invokeNagman(NIL$Lisp,
        "f02aff",
        ["rr"::S,"ri"::S,"a"::S,"integer"::S]$Lisp,
        ["double"::S, ["rr"::S, "n"::S]$Lisp, ["ri"::S, "n"::S]$Lisp,
        ["a"::S, "ia"::S, "n"::S]$Lisp]$Lisp,$Lisp,
        ["integer"::S, ["ia"::S, "n"::S,$Lisp,
         "ifail"::S]$Lisp]$Lisp,
    pretend List (Record(key:Symbol, entry:Any)))$Result

f02ajf(iarArg:Integer,iaiArg:Integer,nArg:Integer,
   iarArg:Matrix DoubleFloat,iaiArg:Matrix DoubleFloat,
   iarArg:Integer): Result ==

   [(invokeNagman(NIL$Lisp,_
      "$f02ajf$",_
       "ar"::S,"ai"::S,"integer"::S$Lisp,_
       ["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_,
       ["integer"::S,"n"::S]$Lisp]$Lisp,$Lisp_]
   )@List Any$Lisp)$Lisp)
   pretend List (Record(key:Symbol,entry:Any))$Result

f02akf(iarArg:Integer,iaiArg:Integer,nArg:Integer,
   iarArg:Matrix DoubleFloat,iaiArg:Matrix DoubleFloat,
   iarArg:Integer): Result ==

   [(invokeNagman(NIL$Lisp,_
      "$f02akf$",_
       "ai"::S,"integer"::S$Lisp,_
       ["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_,
       ["integer"::S,"n"::S]$Lisp]$Lisp]$Lisp)
   )@List Any$Lisp)$Lisp)
   pretend List (Record(key:Symbol,entry:Any))$Result
CHAPTER 15. \textsc{Chapter N}

\$Lisp$
,
]$Lisp,$Lisp


[\{[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any, 
 ifailArg::Any,arArg::Any,aiArg::Any ]_ 
 @List Any|$Lisp)(Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

f02awf(iarArg:Integer,iaiArg:Integer,nArg:Integer, 
 arArg:Matrix DoubleFloat,aiArg:Matrix DoubleFloat, 
 ifailArg:Integer): Result == 
([(invokeNagman(NIL$Lisp, _
"f02awf", 
 "wk1"::S,"wk2"::S_, 
 ,"wk3"::S]$Lisp, 

[\{[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any, 
 ifailArg::Any,arArg::Any,aiArg::Any ]_ 
 @List Any|$Lisp)(Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

f02axf(arArg:Matrix DoubleFloat,iarArg:Integer,aiArg:Matrix DoubleFloat, 
 iaiArg:Integer,nArg:Integer,ivrArg:Integer, 
 iviArg:Integer,ifailArg:Integer): Result == 
([(invokeNagman(NIL$Lisp, _
"f02axf", 
 ,"wk1"::S,"wk2"::S,"wk3"::S]$Lisp, 

[\{[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any, 
 ifailArg::Any ]_ 
 @List Any|$Lisp)(Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

f02axf(arArg:Matrix DoubleFloat,iarArg:Integer,aiArg:Matrix DoubleFloat, 
 iaiArg:Integer,nArg:Integer,ivrArg:Integer, 
 iviArg:Integer,ifailArg:Integer): Result == 
([(invokeNagman(NIL$Lisp, _
"f02axf", 
 ,"wk1"::S,"wk2"::S,"wk3"::S]$Lisp, 

[\{[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any, 
 ifailArg::Any ]_ 
 @List Any|$Lisp)(Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result


```

```

```
"ifail"::S$Lisp,_
[(nArg:: Any, iaArg:: Any, ibArg:: Any, eps1Arg:: Any, matvArg:: Any,_
 ivArg:: Any, ifailArg:: Any, aArg:: Any, bArg:: Any )]
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f02fjf(nArg:Integer,kArg:Integer,tolArg:DoubleFloat,_
novecsArg:Integer,nrxArg:Integer,lworkArg:Integer,_
lrworkArg:Integer,mArg:Integer,_
noeitsArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
dotArg:Union(fn:FileName,fp:Asp27(DOT)),_
imageArg:Union(fn:FileName,fp:Asp28(IMAGE))): Result ==
pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
if dotArg case fn
then outputAsFortran(dotArg.fn)
else outputAsFortran(dotArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
if imageArg case fn
then outputAsFortran(imageArg.fn)
else outputAsFortran(imageArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran()$Asp29(MONIT)
popFortranOutputStack()$FOP
[[invokeNagman([dotFilename,imageFilename,monitFilename]$Lisp,_
"lwork"::S,"lrwork"::S,"m"::S,"noiits"::S_,
"work"::S,"rwork"::S,"iwork"::S_]
$Lisp_,
 "moni"::S])$Lisp_]
$Lisp_,
["iwork"::S,"lwork"::S]$Lisp]$Lisp
]
$Lisp_,
[[nArg:: Any,kArg:: Any,tolArg:: Any,novecsArg:: Any,nrxArg:: Any,_
 lworkArg:: Any,lrworkArg:: Any,iworkArg:: Any,mArg:: Any,_
noeitsArg:: Any,ifailArg:: Any,xArg:: Any ]]
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f02fjf(nArg:Integer,kArg:Integer,tolArg:DoubleFloat,_
novecsArg: Integer, nrxArg: Integer, lworkArg: Integer, 
lrworkArg: Integer, liworkArg: Integer, mArg: Integer, 
noitsArg: Integer, xArg: Matrix DoubleFloat, ifailArg: Integer, 
dotArg: Union(fn: FileName, fp: Asp27(DOT)), 
imageArg: Union(fn: FileName, fp: Asp28(IMAGE)), 
monitArg: FileName): Result ==
pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
if dotArg case fn 
  then outputAsFortran(dotArg.fn)
  else outputAsFortran(dotArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
if imageArg case fn 
  then outputAsFortran(imageArg.fn)
  else outputAsFortran(imageArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran(monitArg)
[(invokeNagman([dotFilename, imageFilename, monitFilename]$Lisp, 
  "f02fjf", 
  "n"::S,"k"::S,"tol"::S,"novecs"::S,"nrx"::S, 
  "lwork"::S,"lrwork"::S,"liwork"::S,"m"::S,"noits"::S, 
  "work"::S,"rwork"::S,"iwork"::S 
 ]$Lisp, 
   "monit"::S]$Lisp, 
 [["double"::S,"tol"::S,"n"::S,"k"::S]$Lisp, 
   ["x"::S,"nrx"::S,"k"::S]$Lisp,["work"::S,"lwork"::S]$Lisp, 
 ]$Lisp, 
 ["iwork"::S,"liwork"::S]$Lisp]$Lisp_ 
 ]$Lisp, 
 ["d"::S,"m"::S,"noits"::S,"x"::S,"ifail"::S]$Lisp, 
 [([nArg::Any,kArg::Any, tolArg::Any, novecsArg::Any, nrxArg::Any, 
   lworkArg::Any, lrworkArg::Any, liworkArg::Any, mArg::Any, 
   noitsArg::Any, ifailArg::Any, xArg::Any ])
]$Lisp
pretend List (Record(key: Symbol, entry: Any))$Result

f02wef(mArg:Integer, nArg:Integer, ldaArg:Integer, 
ncolbArg:Integer, ldbArg:Integer, wantqArg:boolean, 
ldqArg:Integer, wantpArg:boolean, ldptArg:Integer, 
alphaArg:Matrix DoubleFloat, betaArg:Matrix DoubleFloat, 
ifailArg:Integer): Result ==
workLength : Integer :=
  mArg >= nArg =>
  wantqArg and wantpArg =>


\begin{verbatim}
max(max(nArg**2 + 5*(nArg - 1),nArg + ncolbArg),4)
wantqArg =>
    max(max(nArg**2 + 4*(nArg - 1),nArg + ncolbArg),4)
wantpArg =>
    zero? ncolbArg => max(3*(nArg - 1),2)
    max(5*(nArg - 1),2)
    zero? ncolbArg => max(2*(nArg - 1),2)
    max(3*(nArg - 1),2)
wantqArg and wantpArg =>
    max(mArg**2 + 5*(mArg - 1),2)
wantqArg =>
    max(3*(mArg - 1),1)
wantpArg =>
    zero? ncolbArg => max(mArg**2+3*(mArg - 1),2)
    max(mArg**2+5*(mArg - 1),2)
    zero? ncolbArg => max(2*(mArg - 1),1)
    max(3*(mArg - 1),1)
[[invokeNagman(NIL$Lisp,_
    "f02wef",_
    ["m"::S,"n"::S,"lda"::S,"ncolb"::S,"ldb"::S_
    ,"b"::S]$Lisp,_
    [["double"::S,["q"::S,"ldq"::S,"m"::S]$Lisp_,
    ["work"::S,workLength]$Lisp,["a"::S,"lda"::S,"n"::S]$Lisp_,
    ["b"::S,"ldb"::S,"ncolb"::S]$Lisp_]$Lisp$_
    ]$Lisp_,
    ["logical"::S,"wantq":S,"wantp":S]$Lisp_]$Lisp_]
$Lisp_,
    ["integer",["m"::S,"n"::S,"lda"::S,"ncolb"::S_
    ["logical","wantq","wantp"]$Lisp_]
$Lisp,_
    [(mArg::Any,nArg::Any,ldaArg::Any,ncolbArg::Any,ldbArg::Any,_
    wantqArg::Any,ldqArg::Any,wantpArg::Any,ldptArg::Any,_
    ifailArg::Any,aArg::Any,bArg::Any ])_
    @List Any]$Lisp]$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
\end{verbatim}
zero? ncolbArg and wantpArg and not wantqArg => 3*tem
not wantpArg =>
    ncolbArg >0 or wantqArg => 3*tem
5*tem
cLen : Integer :=
mArg => nArg =>
    wantqArg and wantpArg => 2*(nArg + max(nArg**2,ncolbArg))
wantqArg and not wantpArg => 2*(nArg + max(nArg**2+nArg,ncolbArg))
2*(nArg + max(nArg,ncolbArg))
wantpArg => 2*(mArg**2 + mArg)
2*mArg
svLength : Integer :=
min(mArg,nArg)
[(invokeNagman(NIL$Lisp,_
    "f02xef",_
    ["m"::S,"n"::S,"lda"::S,"ncolb"::S,"ldb"::S,_
        "b"::S,"cwork"::S]$Lisp,_
    [["double"::S,"sv"::S,svLength]$Lisp,["rwork"::S,rLen]$Lisp_]
$Lisp_,
    [["integer"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S,_
    ["logical"::S,"wantq"::S,"wantp"::S]$Lisp_]
$Lisp_,
    [["double complex"::S,"q"::S,"ldq"::S,"m"::S]$Lisp_,
$Lisp_,
    [[mArg::Any,nArg::Any,ldaArg::Any,ncolbArg::Any,ldbArg::Any,_
        wantqArg::Any,ldqArg::Any,ncolbArg::Any,_
        ifailArg::Any,aArg::Any,bArg::Any j]_]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any)))$Result
package NAGE02 NagFittingPackage

--- NagFittingPackage.input ---

)set break resume
)sys rm -f NagFittingPackage.output
)spool NagFittingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagFittingPackage
--E 1

)spool
)lisp (bye)

--- NagFittingPackage.help ---

This package uses the NAG Library to find a function which approximates a set of data points. Typically the data contain random errors, as of experimental measurement, which need to be smoothed out. To seek an approximation to the data, it is first necessary to specify for the approximating function a mathematical form (a polynomial, for example) which contains a number of unspecified coefficients: the appropriate fitting routine then derives for the coefficients the values which provide the best fit of that particular form. The package deals mainly with curve and surface fitting (i.e., fitting with functions of one and of two variables) when a polynomial or a cubic spline is used as the fitting function, since these cover the most common needs. However, fitting with other functions and/or more variables can be undertaken by means of general linear or nonlinear routines (some of which are contained in other packages) depending on whether the coefficients in the function occur linearly or nonlinearly. Cases where a graph rather than a set of data points is given can be treated simply by first reading a suitable set of points from the graph. The package also contains routines for evaluating, differentiating and integrating polynomial and spline curves and surfaces, once the numerical values of their coefficients have been determined.
Contents of this Introduction:

1. Scope of the Chapter

2. Background to the Problems
   2.1. Preliminary Considerations
      2.1.1. Fitting criteria: norms
      2.1.2. Weighting of data points
   2.2. Curve Fitting
      2.2.1. Representation of polynomials
      2.2.2. Representation of cubic splines
   2.3. Surface Fitting
      2.3.1. Bicubic splines: definition and representation
   2.4. General Linear and Nonlinear Fitting Functions
   2.5. Constrained Problems
   2.6. References

3. Recommendations on Choice and Use of Routines
   3.1. General
      3.1.1. Data considerations
      3.1.2. Transformation of variables
   3.2. Polynomial Curves
      3.2.1. Least-squares polynomials: arbitrary data points
      3.2.2. Least-squares polynomials: selected data points
   3.3. Cubic Spline Curves
      3.3.1. Least-squares cubic splines
      3.3.2. Automatic fitting with cubic splines
   3.4. Spline Surfaces
1. Scope of the Chapter

The main aim of this chapter is to assist the user in finding a function which approximates a set of data points. Typically the data contain random errors, as of experimental measurement, which need to be smoothed out. To seek an approximation to the data, it is first necessary to specify for the approximating function a mathematical form (a polynomial, for example) which contains a number of unspecified coefficients: the appropriate fitting routine then derives for the coefficients the values which provide the best fit of that particular form. The chapter deals mainly with curve and surface fitting (i.e., fitting with functions of one and of two variables) when a polynomial or a cubic spline is used as the fitting function, since these cover the most common needs. However, fitting with other functions and/or more variables can be undertaken by means of general linear or nonlinear routines (some of which are contained in other chapters) depending on whether the coefficients in the function occur linearly or nonlinearly. Cases where a graph rather than a set of data points is given can be treated simply by first reading a suitable set of points from the graph.

The chapter also contains routines for evaluating, differentiating and integrating polynomial and spline curves and surfaces, once the numerical values of their coefficients have been determined.

2. Background to the Problems

2.1. Preliminary Considerations

In the curve-fitting problems considered in this chapter, we have
a dependent variable $y$ and an independent variable $x$, and we are
given a set of data points $(x_r, y_r)$, for $r=1,2,\ldots,m$. The
preliminary matters to be considered in this section will, for
simplicity, be discussed in this context of curve-fitting
problems. In fact, however, these considerations apply equally
well to surface and higher-dimensional problems. Indeed, the
discussion presented carries over essentially as it stands if,
for these cases, we interpret $x$ as a vector of several
independent variables and correspondingly each $x_r$ as a vector
containing the $r$th data value of each independent variable.

We wish, then, to approximate the set of data points as closely
as possible with a specified function, $f(x)$ say, which is as
smooth as possible -- $f(x)$ may, for example, be a polynomial. The
requirements of smoothness and closeness conflict, however, and a
balance has to be struck between them. Most often, the smoothness
requirement is met simply by limiting the number of coefficients
allowed in the fitting function -- for example, by restricting
the degree in the case of a polynomial. Given a particular number
of coefficients in the function in question, the fitting routines
of this chapter determine the values of the coefficients such
that the 'distance' of the function from the data points is as
small as possible. The necessary balance is struck by the user
comparing a selection of such fits having different numbers of
coefficients. If the number of coefficients is too low, the
approximation to the data will be poor. If the number is too
high, the fit will be too close to the data, essentially
following the random errors and tending to have unwanted
fluctuations between the data points. Between these extremes,
there is often a group of fits all similarly close to the data
points and then, particularly when least-squares polynomials are
used, the choice is clear: it is the fit from this group having
the smallest number of coefficients.

The above process can be seen as the user minimizing the
smoothness measure (i.e., the number of coefficients) subject to
the distance from the data points being acceptably small. Some of
the routines, however, do this task themselves. They use a
different measure of smoothness (in each case one that is
continuous) and minimize it subject to the distance being less
than a threshold specified by the user. This is a much more
automatic process, requiring only some experimentation with the
threshold.

2.1.1. Fitting criteria: norms

A measure of the above 'distance' between the set of data points
and the function $f(x)$ is needed. The distance from a single data
point \((x, y)\) to the function can simply be taken as

\[
(\epsilon_r) = y - f(x),
\]

and is called the residual of the point. (With this definition, the residual is regarded as a function of the coefficients contained in \(f(x)\); however, the term is also used to mean the particular value of \(\epsilon_r\) which corresponds to the fitted \(r\) values of the coefficients.) However, we need a measure of distance for the set of data points as a whole. Three different measures are used in the different routines (which measure to select, according to circumstances, is discussed later in this sub-section). With \(\epsilon_r\) defined in (1), these measures, or \(r\) norms, are

\[
\begin{align*}
&\sum_{r=1}^{m} |\epsilon_r|, \quad (2) \\
&\sqrt{\sum_{r=1}^{m} \epsilon_r^2}, \quad (3) \\
&\max_{1 \leq r \leq m} |\epsilon_r|, \quad (4)
\end{align*}
\]

respectively the \(l_1\) norm, the \(l_2\) norm and the \(l_{\infty}\) norm.

Minimization of one or other of these norms usually provides the fitting criterion, the minimization being carried out with respect to the coefficients in the mathematical form used for \(f(x)\): with respect to the \(b\) for example if the mathematical form \(\sum_{i=1}^{n} a_i x^i\) is the power series in (8) below. The fit which results from minimizing (2) is known as the \(l_1\) fit, or the fit in the \(l_1\) norm: that which results from minimizing (3) is the \(l_2\) fit, the well-known least-squares fit (minimizing (3) is equivalent to
minimizing the square of (3), i.e., the sum of squares of residuals, and it is the latter which is used in practice), and that from minimizing (4) is the $l_\infty$, or minimax, fit.

Strictly speaking, implicit in the use of the above norms are the statistical assumptions that the random errors in the $y$ are independent of one another and that any errors in the $x$ are negligible by comparison. From this point of view, the use of the $l_1$ norm is appropriate when the random errors in the $y$ have a normal distribution, and the $l_2$ norm is appropriate when they have a rectangular distribution, as when fitting a table of values rounded to a fixed number of decimal places. The $l_1$ norm is appropriate when the error distribution has its frequency function proportional to the negative exponential of the modulus of the normalised error -- not a common situation.

However, the user is often indifferent to these statistical considerations, and simply seeks a fit which he can assess by inspection, perhaps visually from a graph of the results. In this event, the $l_1$ norm is particularly appropriate when the data are thought to contain some 'wild' points (since fitting in this norm tends to be unaffected by the presence of a small number of such points), though of course in simple situations the user may prefer to identify and reject these points. The $l_\infty$ norm should be used only when the maximum residual is of particular concern, as may be the case for example when the data values have been obtained by accurate computation, as of a mathematical function. Generally, however, a routine based on least-squares should be preferred, as being computationally faster and usually providing more information on which to assess the results. In many problems the three fits will not differ significantly for practical purposes.

Some of the routines based on the $l_1$ norm do not minimize the $l_2$ norm itself but instead minimize some (intuitively acceptable) measure of smoothness subject to the norm being less than a user-specified threshold. These routines fit with cubic or bicubic splines (see (10) and (14) below) and the smoothing measures relate to the size of the discontinuities in their third derivatives. A much more automatic fitting procedure follows from this approach.
2.1.2. Weighting of data points

The use of the above norms also assumes that the data values $y^r$ are of equal (absolute) accuracy. Some of the routines enable an allowance to be made to take account of differing accuracies. The allowance takes the form of 'weights' applied to the $y$-values so that those values known to be more accurate have a greater influence on the fit than others. These weights, to be supplied by the user, should be calculated from estimates of the absolute accuracies of the $y$-values, these estimates being expressed as standard deviations, probable errors or some other measure which has the same dimensions as $y$. Specifically, for each $y$ the corresponding weight $w^r$ should be inversely proportional to the accuracy estimate of $y^r$. For example, if the percentage accuracy $r$ is the same for all $y$, then the absolute accuracy of $y$ is proportional to $y^r$ (assuming $y$ to be positive, as it usually is in such cases) and so $w^r = K/y^r$, for $r=1,2,\ldots,m$, for an arbitrary positive constant $K$. (This definition of weight is stressed because often weight is defined as the square of that used here.) The norms (2), (3) and (4) above are then replaced respectively by

$$\sum_{r=1}^{m} \left| w^r (\epsilon) \right|, \quad (5)$$

$$\sum_{r=1}^{m} w^r (\epsilon)^2, \quad (6)$$

$$\max_{r} \left| w^r (\epsilon) \right|. \quad (7)$$

Again it is the square of (6) which is used in practice rather than (6) itself.
2.2. Curve Fitting

When, as is commonly the case, the mathematical form of the fitting function is immaterial to the problem, polynomials and cubic splines are to be preferred because their simplicity and ease of handling confer substantial benefits. The cubic spline is the more versatile of the two. It consists of a number of cubic polynomial segments joined end to end with continuity in first and second derivatives at the joins. The third derivative at the joins is in general discontinuous. The x-values of the joins are called knots, or, more precisely, interior knots. Their number determines the number of coefficients in the spline, just as the degree determines the number of coefficients in a polynomial.

2.2.1. Representation of polynomials

Rather than using the power-series form

\[ f(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_k x^k \]  

(8)

to represent a polynomial, the routines in this chapter use the Chebyshev series form

\[ f(x) = -a_0 T_0(x) + a_1 T_1(x) + a_2 T_2(x) + \ldots + a_k T_k(x), \]  

(9)

where \( T_i(x) \) is the Chebyshev polynomial of the first kind of degree \( i \) (see Cox and Hayes [1], page 9), and where the range of \( x \) has been normalised to run from \(-1\) to \(+1\). The use of either form leads theoretically to the same fitted polynomial, but in practice results may differ substantially because of the effects of rounding error. The Chebyshev form is to be preferred, since it leads to much better accuracy in general, both in the computation of the coefficients and in the subsequent evaluation of the fitted polynomial at specified points. This form also has other advantages: for example, since the later terms in (9) generally decrease much more rapidly from left to right than do those in (8), the situation is more often encountered where the last terms are negligible and it is obvious that the degree of the polynomial can be reduced (note that on the interval \(-1 \leq x \leq 1\) for all \( i \), \( T_i(x) \) attains the value unity but never exceeds it, so that the coefficient \( a_i \) gives directly the maximum value of the \( i \)th term containing it).
2.2.2. Representation of cubic splines

A cubic spline is represented in the form

\[ f(x) = c_1 N_1(x) + c_2 N_2(x) + \ldots + c_p N_p(x), \]  

where \( N_i(x) \), for \( i = 1, 2, \ldots, p \), is a normalised cubic B-spline (see Hayes [2]). This form, also, has advantages of computational speed and accuracy over alternative representations.

2.3. Surface Fitting

There are now two independent variables, and we shall denote these by \( x \) and \( y \). The dependent variable, which was denoted by \( y \) in the curve-fitting case, will now be denoted by \( f \). (This is a rather different notation from that indicated for the general-dimensional problem in the first paragraph of Section 2.1, but it has some advantages in presentation.)

Again, in the absence of contrary indications in the particular application being considered, polynomials and splines are the approximating functions most commonly used. Only splines are used by the surface-fitting routines in this chapter.

2.3.1. Bicubic splines: definition and representation

The bicubic spline is defined over a rectangle \( R \) in the \((x,y)\) plane, the sides of \( R \) being parallel to the \( x \)- and \( y \)-axes. \( R \) is divided into rectangular panels, again by lines parallel to the axes. Over each panel the bicubic spline is a bicubic polynomial, that is it takes the form

\[ a_{ij} x^i y^j. \]  

Each of these polynomials joins the polynomials in adjacent panels with continuity up to the second derivative. The constant \( x \)-values of the dividing lines parallel to the \( y \)-axis form the set of interior knots for the variable \( x \), corresponding precisely to the set of interior knots of a cubic spline. Similarly, the constant \( y \)-values of dividing lines parallel to the \( x \)-axis form the set of interior knots for the variable \( y \). Instead of representing the bicubic spline in terms of the above set of bicubic polynomials, however, it is represented, for the sake of computational speed and accuracy, in the form
\[ f(x,y) = \sum_{i=1}^{p} \sum_{j=1}^{q} c_{ij} M_i(x) N_j(y), \quad (14) \]

where \( M_i(x) \), for \( i=1,2,...,p \), and \( N_j(y) \), for \( j=1,2,...,q \), are normalised B-splines (see Hayes and Halliday [4] for further details of bicubic splines and Hayes [2] for normalised B-splines).

2.4. General Linear and Nonlinear Fitting Functions

We have indicated earlier that, unless the data-fitting application under consideration specifically requires some other type of fitting function, a polynomial or a spline is usually to be preferred. Special routines for these functions, in one and in two variables, are provided in this chapter. When the application does specify some other fitting function, however, it may be treated by a routine which deals with a general linear function, or by one for a general nonlinear function, depending on whether the coefficients in the given function occur linearly or nonlinearly.

The general linear fitting function can be written in the form

\[ f(x) = c_1 \phi_1(x) + c_2 \phi_2(x) + \ldots + c_p \phi_p(x), \quad (15) \]

where \( x \) is a vector of one or more independent variables, and the \( \phi_i \) are any given functions of these variables (though they must be linearly independent of one another if there is to be the possibility of a unique solution to the fitting problem). This is not intended to imply that each \( \phi_i \) is necessarily a function of all the variables: we may have, for example, that each \( \phi_i \) is a function of a different single variable, and even that one of the \( \phi_i \) is a constant. All that is required is that a value of each \( \phi_i(x) \) can be computed when a value of each independent variable is given.

When the fitting function \( f(x) \) is not linear in its coefficients, no more specific representation is available in general than \( f(x) \) itself. However, we shall find it helpful later on to indicate
the fact that \( f(x) \) contains a number of coefficients (to be determined by the fitting process) by using instead the notation \( f(x;c) \), where \( c \) denotes the vector of coefficients. An example of a nonlinear fitting function is

\[
f(x;c) = c_1 + c_2 \exp(-c_3 x) + c_4 \exp(-c_5 x),
\]

which is in one variable and contains five coefficients. Note that here, as elsewhere in this Chapter Introduction, we use the term 'coefficients' to include all the quantities whose values are to be determined by the fitting process, not just those which occur linearly. We may observe that it is only the presence of the coefficients \( c_4 \) and \( c_5 \) which makes the form (16) nonlinear.

If the values of these two coefficients were known beforehand, (16) would instead be a linear function which, in terms of the general linear form (15), has \( p=3 \) and

\[
(\phi^i(x)) = \begin{cases} 1, & \phi^i(x) = \exp(-c_3 x), \\ \exp(-c_5 x). & \end{cases}
\]

We may note also that polynomials and splines, such as (9) and (14), are themselves linear in their coefficients. Thus if, when fitting with these functions, a suitable special routine is not available (as when more than two independent variables are involved or when fitting in the \( l_1 \) norm), it is appropriate to use a routine designed for a general linear function.

2.5. Constrained Problems

So far, we have considered only fitting processes in which the values of the coefficients in the fitting function are determined by an unconstrained minimization of a particular norm. Some fitting problems, however, require that further restrictions be placed on the determination of the coefficient values. Sometimes these restrictions are contained explicitly in the formulation of the problem in the form of equalities or inequalities which the coefficients, or some function of them, must satisfy. For example, if the fitting function contains a term \( A \exp(-kx) \), it may be required that \( k \geq 0 \). Often, however, the equality or inequality constraints relate to the value of the fitting function or its derivatives at specified values of the independent variable(s), but these too can be expressed in terms of the coefficients of the fitting function, and it is appropriate to do this if a general linear or nonlinear routine is being used. For example, if the fitting function is that given in (10), the requirement that the first derivative of the
function at \( x=x_0 \) be non-negative can be expressed as
\[
\frac{d^3}{dx^3} f(x) + \frac{d^2}{dx^2} f(x) + \frac{d}{dx} f(x) + f(x) \geq 0,
\]
where the prime denotes differentiation with respect to \( x \) and each derivative is evaluated at \( x=x_0 \). On the other hand, if the requirement had been that the derivative at \( x=x_0 \) be exactly zero, the inequality sign in (17) would be replaced by an equality.

Routines which provide a facility for minimizing the appropriate norm subject to such constraints are discussed in Section 3.6.

2.6. References


(For definition of normalised B-splines and details of numerical methods.)


3. Recommendations on Choice and Use of Routines

3.1. General

The choice of a routine to treat a particular fitting problem will depend first of all on the fitting function and the norm to be used. Unless there is good reason to the contrary, the fitting function should be a polynomial or a cubic spline (in the appropriate number of variables) and the norm should be the \( L_2 \) norm (leading to the least-squares fit). If some other function is to be used, the choice of routine will depend on whether the function is nonlinear (in which case see Section 3.5.2) or linear in its coefficients (see Section 3.5.1), and, in the latter case,
on whether the $1$ or $1^2$ norm is to be used. The latter section is appropriate for polynomials and splines, too, if the $1$ norm is preferred.

In the case of a polynomial or cubic spline, if there is only one independent variable, the user should choose a spline (Section 3.3) when the curve represented by the data is of complicated form, perhaps with several peaks and troughs. When the curve is of simple form, first try a polynomial (see Section 3.2) of low degree, say up to degree 5 or 6, and then a spline if the polynomial fails to provide a satisfactory fit. (Of course, if third-derivative discontinuities are unacceptable to the user, a polynomial is the only choice.) If the problem is one of surface fitting, one of the spline routines should be used (Section 3.4). If the problem has more than two independent variables, it may be treated by the general linear routine in Section 3.5.1, again using a polynomial in the first instance.

Another factor which affects the choice of routine is the presence of constraints, as previously discussed in Section 2.5. Indeed this factor is likely to be overriding at present, because of the limited number of routines which have the necessary facility. See Section 3.6.

3.1.1. Data considerations

A satisfactory fit cannot be expected by any means if the number and arrangement of the data points do not adequately represent the character of the underlying relationship: sharp changes in behaviour, in particular, such as sharp peaks, should be well covered. Data points should extend over the whole range of interest of the independent variable(s); extrapolation outside the data ranges is most unwise. Then, with polynomials, it is advantageous to have additional points near the ends of the ranges, to counteract the tendency of polynomials to develop fluctuations in these regions. When, with polynomial curves, the user can precisely choose the $x$-values of the data, the special points defined in Section 3.2.2 should be selected. With splines the choice is less critical as long as the character of the relationship is adequately represented. All fits should be tested graphically before accepting them as satisfactory.

For this purpose it should be noted that it is not sufficient to plot the values of the fitted function only at the data values of the independent variable(s); at the least, its values at a similar number of intermediate points should also be plotted, as unwanted fluctuations may otherwise go undetected. Such fluctuations are the less likely to occur the lower the number of
coefficients chosen in the fitting function. No firm guide can be given, but as a rough rule, at least initially, the number of coefficients should not exceed half the number of data points (points with equal or nearly equal values of the independent variable, or both independent variables in surface fitting, counting as a single point for this purpose). However, the situation may be such, particularly with a small number of data points, that a satisfactorily close fit to the data cannot be achieved without unwanted fluctuations occurring. In such cases, it is often possible to improve the situation by a transformation of one or more of the variables, as discussed in the next paragraph: otherwise it will be necessary to provide extra data points. Further advice on curve fitting is given in Cox and Hayes [1] and, for polynomials only, in Hayes [3] of Section 2.7. Much of the advice applies also to surface fitting; see also the Routine Documents.

3.1.2. Transformation of variables

Before starting the fitting, consideration should be given to the choice of a good form in which to deal with each of the variables: often it will be satisfactory to use the variables as they stand, but sometimes the use of the logarithm, square root, or some other function of a variable will lead to a better-behaved relationship. This question is customarily taken into account in preparing graphs and tables of a relationship and the same considerations apply when curve or surface fitting. The practical context will often give a guide. In general, it is best to avoid having to deal with a relationship whose behaviour in one region is radically different from that in another. A steep rise at the left-hand end of a curve, for example, can often best be treated by curve fitting in terms of log(x+c) with some suitable value of the constant c. A case when such a transformation gave substantial benefit is discussed in Hayes [3] page 60. According to the features exhibited in any particular case, transformation of either dependent variable or independent variable(s) or both may be beneficial. When there is a choice it is usually better to transform the independent variable(s): if the dependent variable is transformed, the weights attached to the data points must be adjusted. Thus (denoting the dependent variable by y, as in the notation for curves) if the y to be fitted have been obtained by a transformation y=g(Y) from original data values Y, with weights W, for r=1,2,...,m, we must take

\[ w = \frac{W}{(dy/dY)} \]

(18)
where the derivative is evaluated at \( Y \). Strictly, the transformation of \( Y \) and the adjustment of weights are valid only when the data errors in the \( Y \) are small compared with the range spanned by the \( Y \), but this is usually the case.

### 3.2. Polynomial Curves

#### 3.2.1. Least-squares polynomials: arbitrary data points

E02ADF fits to arbitrary data points, with arbitrary weights, polynomials of all degrees up to a maximum degree \( k \), which is at choice. If the user is seeking only a low degree polynomial, up to degree 5 or 6 say, \( k=10 \) is an appropriate value, providing there are about 20 data points or more. To assist in deciding the degree of polynomial which satisfactorily fits the data, the routine provides the root-mean-square-residual \( s_i \) for all degrees \( i=1,2,...,k \). In a satisfactory case, these \( s_i \) will decrease steadily as \( i \) increases and then settle down to a fairly constant value, as shown in the example

\[
\begin{array}{c|c}
  i & s_i \\
 0 & 3.5215 \\
 1 & 0.7708 \\
 2 & 0.1861 \\
 3 & 0.0820 \\
 4 & 0.0554 \\
 5 & 0.0251 \\
 6 & 0.0264 \\
 7 & 0.0280 \\
 8 & 0.0277 \\
 9 & 0.0297 \\
 10 & 0.0271 \\
\end{array}
\]

If the $s_i$ values settle down in this way, it indicates that the closest polynomial approximation justified by the data has been achieved. The degree which first gives the approximately constant value of $s_i$ (degree 5 in the example) is the appropriate degree to select. (Users who are prepared to accept a fit higher than sixth degree, should simply find a high enough value of $k$ to enable the type of behaviour indicated by the example to be detected: thus they should seek values of $k$ for which at least 4 or 5 consecutive values of $s_i$ are approximately the same.) If the degree were allowed to go high enough, $s_i$ would, in most cases, eventually start to decrease again, indicating that the data points are being fitted too closely and that undesirable fluctuations are developing between the points. In some cases, particularly with a small number of data points, this final decrease is not distinguishable from the initial decrease in $s_i$.

In such cases, users may seek an acceptable fit by examining the graphs of several of the polynomials obtained. Failing this, they may (a) seek a transformation of variables which improves the behaviour, (b) try fitting a spline, or (c) provide more data points. If data can be provided simply by drawing an approximating curve by hand and reading points from it, use the points discussed in Section 3.2.2.

3.2.2. Least-squares polynomials: selected data points

When users are at liberty to choose the $x$-values of data points, such as when the points are taken from a graph, it is most advantageous when fitting with polynomials to use the values $x = \cos((\pi)r/n)$, for $r=0,1,\ldots,n$ for some value of $n$, a suitable value for which is discussed at the end of this section. Note that these $x$ relate to the variable $x$ after it has been normalised so that its range of interest is $-1$ to $+1$. E02ADF may then be used as in Section 3.2.1 to seek a satisfactory fit.

3.3. Cubic Spline Curves

3.3.1. Least-squares cubic splines

E02BAF fits to arbitrary data points, with arbitrary weights, a cubic spline with interior knots specified by the user. The choice of these knots so as to give an acceptable fit must largely be a matter of trial and error, though with a little experience a satisfactory choice can often be made after one or
two trials. It is usually best to start with a small number of knots (too many will result in unwanted fluctuations in the fit, or even in there being no unique solution) and, examining the fit graphically at each stage, to add a few knots at a time at places where the fit is particularly poor. Moving the existing knots towards these places will also often improve the fit. In regions where the behaviour of the curve underlying the data is changing rapidly, closer knots will be needed than elsewhere. Otherwise, positioning is not usually very critical and equally-spaced knots are often satisfactory. See also the next section, however.

A useful feature of the routine is that it can be used in applications which require the continuity to be less than the normal continuity of the cubic spline. For example, the fit may be required to have a discontinuous slope at some point in the range. This can be achieved by placing three coincident knots at the given point. Similarly a discontinuity in the second derivative at a point can be achieved by placing two knots there. Analogy with these discontinuous cases can provide guidance in more usual cases: for example, just as three coincident knots can produce a discontinuity in slope, so three close knots can produce a rapid change in slope. The closer the knots are, the more rapid can the change be.

Figure 1
Please see figure in printed Reference Manual

An example set of data is given in Figure 1. It is a rather tricky set, because of the scarcity of data on the right, but it will serve to illustrate some of the above points and to show some of the dangers to be avoided. Three interior knots (indicated by the vertical lines at the top of the diagram) are chosen as a start. We see that the resulting curve is not steep enough in the middle and fluctuates at both ends, severely on the right. The spline is unable to cope with the shape and more knots are needed.

In Figure 2, three knots have been added in the centre, where the data shows a rapid change in behaviour, and one further out at each end, where the fit is poor. The fit is still poor, so a further knot is added in this region and, in Figure 3, disaster ensues in rather spectacular fashion.

Figure 2
Please see figure in printed Reference Manual

Figure 3
Please see figure in printed Reference Manual

The reason is that, at the right-hand end, the fits in Figure 1
and Figure 2 have been interpreted as poor simply because of the fluctuations about the curve underlying the data (or what it is naturally assumed to be). But the fitting process knows only about the data and nothing else about the underlying curve, so it is important to consider only closeness to the data when deciding goodness of fit.

Thus, in Figure 1, the curve fits the last two data points quite well compared with the fit elsewhere, so no knot should have been added in this region. In Figure 2, the curve goes exactly through the last two points, so a further knot is certainly not needed here.

Figure 4
Please see figure in printed Reference Manual

Figure 4 shows what can be achieved without the extra knot on each of the flat regions. Remembering that within each knot interval the spline is a cubic polynomial, there is really no need to have more than one knot interval covering each flat region.

What we have, in fact, in Figure 2 and Figure 3 is a case of too many knots (so too many coefficients in the spline equation) for the number of data points. The warning in the second paragraph of Section 2.1 was that the fit will then be too close to the data, tending to have unwanted fluctuations between the data points. The warning applies locally for splines, in the sense that, in localities where there are plenty of data points, there can be a lot of knots, as long as there are few knots where there are few points, especially near the ends of the interval. In the present example, with so few data points on the right, just the one extra knot in Figure 2 is too many! The signs are clearly present, with the last two points fitted exactly (at least to the graphical accuracy and actually much closer than that) and fluctuations within the last two knot-intervals (cf. Figure 1, where only the final point is fitted exactly and one of the wobbles spans several data points).

The situation in Figure 3 is different. The fit, if computed exactly, would still pass through the last two data points, with even more violent fluctuations. However, the problem has become so ill-conditioned that all accuracy has been lost. Indeed, if the last interior knot were moved a tiny amount to the right, there would be no unique solution and an error message would have been caused. Near-singularity is, sadly, not picked up by the routine, but can be spotted readily in a graph, as Figure 3. B-spline coefficients becoming large, with alternating signs, is another indication. However, it is better to avoid such
situations, firstly by providing, whenever possible, data adequately covering the range of interest, and secondly by placing knots only where there is a reasonable amount of data.

The example here could, in fact, have utilised from the start the observation made in the second paragraph of this section, that three close knots can produce a rapid change in slope. The example has two such rapid changes and so requires two sets of three close knots (in fact, the two sets can be so close that one knot can serve in both sets, so only five knots prove sufficient in Figure 4). It should be noted, however, that the rapid turn occurs within the range spanned by the three knots. This is the reason that the six knots in Figure 2 are not satisfactory as they do not quite span the two turns.

Some more examples to illustrate the choice of knots are given in Cox and Hayes [1].

3.3.2. Automatic fitting with cubic splines

E02BEF also fits cubic splines to arbitrary data points with arbitrary weights but itself chooses the number and positions of the knots. The user has to supply only a threshold for the sum of squares of residuals. The routine first builds up a knot set by a series of trial fits in the $l^2$ norm. Then, with the knot set decided, the final spline is computed to minimize a certain smoothing measure subject to satisfaction of the chosen threshold. Thus it is easier to use than E02BAF (see previous section), requiring only some experimentation with this threshold. It should therefore be first choice unless the user has a preference for the ordinary least-squares fit or, for example, wishes to experiment with knot positions, trying to keep their number down (E02BEF aims only to be reasonably frugal with knots).

3.4. Spline Surfaces

3.4.1. Least-squares bicubic splines

E02DAF fits to arbitrary data points, with arbitrary weights, a bicubic spline with its two sets of interior knots specified by the user. For choosing these knots, the advice given for cubic splines, in Section 3.3.1 above, applies here too. (See also the next section, however.) If changes in the behaviour of the surface underlying the data are more marked in the direction of one variable than of the other, more knots will be needed for the former variable than the latter. Note also that, in the surface case, the reduction in continuity caused by coincident knots will extend across the whole spline surface: for example, if three
knots associated with the variable \( x \) are chosen to coincide at a value \( L \), the spline surface will have a discontinuous slope across the whole extent of the line \( x=L \).

With some sets of data and some choices of knots, the least-squares bicubic spline will not be unique. This will not occur, with a reasonable choice of knots, if the rectangle \( R \) is well covered with data points: here \( R \) is defined as the smallest rectangle in the \((x,y)\) plane, with sides parallel to the axes, which contains all the data points. Where the least-squares solution is not unique, the minimal least-squares solution is computed, namely that least-squares solution which has the smallest value of the sum of squares of the \( B \)-spline coefficients \( c \) (see the end of Section 2.3.2 above). This choice of least-squares solution tends to minimize the risk of unwanted fluctuations in the fit. The fit will not be reliable, however, in regions where there are few or no data points.

3.4.2. Automatic fitting with bicubic splines

E02DDF also fits bicubic splines to arbitrary data points with arbitrary weights but chooses the knot sets itself. The user has to supply only a threshold for the sum of squares of residuals. Just like the automatic curve E02BEF (Section 3.3.2), E02DDF then builds up the knot sets and finally fits a spline minimizing a smoothing measure subject to satisfaction of the threshold. Again, this easier to use routine is normally to be preferred, at least in the first instance.

E02DCF is a very similar routine to E02DDF but deals with data points of equal weight which lie on a rectangular mesh in the \((x,y)\) plane. This kind of data allows a very much faster computation and so is to be preferred when applicable. Substantial departures from equal weighting can be ignored if the user is not concerned with statistical questions, though the quality of the fit will suffer if this is taken too far. In such cases, the user should revert to E02DDF.

3.5. General Linear and Nonlinear Fitting Functions

3.5.1. General linear functions

For the general linear function (15), routines are available for fitting in the \( l_1 \) and \( l_2 \) norms. The least-squares routines (which are to be preferred unless there is good reason to use another norm -- see Section 2.1.1) are in Chapter F04. The \( l_1 \) routine is E02GAF.
All the above routines are essentially linear algebra routines, and in considering their use we need to view the fitting process in a slightly different way from hitherto. Taking \( y \) to be the dependent variable and \( x \) the vector of independent variables, we have, as for equation (1) but with each \( x \) now a vector,

\[
(\epsilon_r) = y - f(x_r) \quad r = 1, 2, \ldots, m. 
\]

Substituting for \( f(x) \) the general linear form (15), we can write this as

\[
c_1(\phi_1(x)) + c_2(\phi_2(x)) + \cdots + c_p(\phi_p(x)) = y - (\epsilon_r),
\]

\[
r = 1, 2, \ldots, m
\]

Thus we have a system of linear equations in the coefficients \( c_j \).

Usually, in writing these equations, the \( \epsilon_r \) are omitted and simply taken as implied. The system of equations is then described as an overdetermined system (since we must have \( m \geq p \) if there is to be the possibility of a unique solution to our fitting problem), and the fitting process of computing the \( c_j \) to minimize one or other of the norms (2), (3) and (4) can be described, in relation to the system of equations, as solving the overdetermined system in that particular norm. In matrix notation, the system can be written as

\[
(\Phi)c = y,
\]

where \( (\Phi) \) is the \( m \) by \( p \) matrix whose element in row \( r \) and column \( j \) is \( (\phi_j(x_r)) \), for \( r = 1, 2, \ldots, m; j = 1, 2, \ldots, p \). The vectors \( c \) and \( y \) respectively contain the coefficients \( c_j \) and the data \( y_r \).

The routines, however, use the standard notation of linear algebra, the overdetermined system of equations being denoted by

\[
Ax = b
\]

The correspondence between this notation and that which we have used for the data-fitting problem (equation (20)) is therefore
given by

$$A = (\Phi), \quad x = c, \quad b = y$$ \hfill (22)

Note that the norms used by these routines are the unweighted norms (2) and (3). If the user wishes to apply weights to the data points, that is to use the norms (5) or (6), the equivalences (22) should be replaced by

$$A = D(\Phi), \quad x = c, \quad b = Dy$$

where $D$ is a diagonal matrix with $w_r$ as the $r$th diagonal element.

Here $w_r$, for $r=1,2,...,m$, is the weight of the $r$th data point as defined in Section 2.1.2.

3.5.2. Nonlinear functions

Routines for fitting with a nonlinear function in the $l^1$ norm are provided in Chapter E04, and that chapter's Introduction should be consulted for the appropriate choice of routine. Again, however, the notation adopted is different from that we have used for data fitting. In the latter, we denote the fitting function by $f(x;c)$, where $x$ is the vector of independent variables and $c$ is the vector of coefficients, whose values are to be determined. The squared $l^2$ norm, to be minimized with respect to the elements of $c$, is then

$$\sum_{r=1}^{m} w_r [y_r - f(x_r; c)]^2$$ \hfill (23)

where $y_r$ is the $r$th data value of the dependent variable, $x_r$ is the vector containing the $r$th values of the independent variables, and $w_r$ is the corresponding weight as defined in Section 2.1.2.

On the other hand, in the nonlinear least-squares routines of Chapter E04, the function to be minimized is denoted by

$$\sum_{r=1}^{m} w_r [y_r - f(x)]$$ \hfill (24)
the minimization being carried out with respect to the elements of the vector \( x \). The correspondence between the two notations is given by

\[
x = c \quad \text{and} \quad f(x) = w[y - f(x ; c)], \quad i = 1, 2, \ldots, m.
\]

Note especially that the vector \( x \) of variables of the nonlinear least-squares routines is the vector \( c \) of coefficients of the data-fitting problem, and in particular that, if the selected routine requires derivatives of the \( f(x) \) to be provided, these are derivatives of \( w[y - f(x ; c)] \) with respect to the coefficients of the data-fitting problem.

### 3.6. Constraints

At present, there are only a limited number of routines which fit subject to constraints. Chapter E04 contains a routine, E04UCF, which can be used for fitting with a nonlinear function in the \( L_2 \) norm subject to equality or inequality constraints. This routine, unlike those in that chapter suited to the unconstrained case, is not designed specifically for minimizing functions which are sums of squares, and so the function (23) has to be treated as a general nonlinear function. The E04 Chapter Introduction should be consulted.

The remaining constraint routine relates to fitting with polynomials in the \( L_2 \) norm. E02AGF deals with polynomial curves and allows precise values of the fitting function and (if required) all its derivatives up to a given order to be prescribed at one or more values of the independent variable.

### 3.7. Evaluation, Differentiation and Integration

Routines are available to evaluate, differentiate and integrate polynomials in Chebyshev-series form and cubic or bicubic splines in B-spline form. These polynomials and splines may have been produced by the various fitting routines or, in the case of polynomials, from prior calls of the differentiation and integration routines themselves.
E02AEF and E02AKF evaluate polynomial curves: the latter has a longer parameter list but does not require the user to normalise the values of the independent variable and can accept coefficients which are not stored in contiguous locations. E02BBF evaluates cubic spline curves, and E02DEF and E02DFF bicubic spline surfaces.

Differentiation and integration of polynomial curves are carried out by E02AHF and E02AJF respectively. The results are provided in Chebyshev-series form and so repeated differentiation and integration are catered for. Values of the derivative or integral can then be computed using the appropriate evaluation routine.

For splines the differentiation and integration routines provided are of a different nature from those for polynomials. E02BCF provides values of a cubic spline curve and its first three derivatives (the rest, of course, are zero) at a given value of $x$ over its whole range. These routines can also be applied to surfaces of the form (14). For example, if, for each value of $j$ in turn, the coefficients $c_{ij}$, for $i=1,2,\ldots,p$ are supplied to E02BCF with $x=x_0$ and on each occasion we select from the output the value of the second derivative, $d_j$ say, and if the whole set $d_j$ are then supplied to the same routine with $x=y_0$, the output will contain all the values at $(x_0,y_0)$ of

$$\frac{2}{2} \frac{dd f}{r+2} \frac{dd f}{2} \frac{d^r f}{dd x ddx ddy} \quad r=1,2,3.$$  

Equally, if after each of the first $p$ calls of E02BCF we had selected the function value (E02BBF would also provide this) instead of the second derivative and we had supplied these values to E02BDF, the result obtained would have been the value of $B$

$$\int_{A}^{B} f(x,y)\,dy,$$

where $A$ and $B$ are the end-points of the $y$ interval over which the spline was defined.
3.8. Index

Automatic fitting,
  with bicubic splines E02DCF
  with cubic splines E02DDF
Data on rectangular mesh E02DCF
Differentiation,
  of cubic splines E02BCF
  of polynomials E02AHF
Evaluation,
  of bicubic splines E02DEF
  of cubic splines E02DF
  of cubic splines and derivatives E02BCF
  of definite integral of cubic splines E02BDF
  of polynomials E02AEF
Integration,
  of cubic splines (definite integral) E02BDF
  of polynomials E02AJF
Least-squares curve fit,
  with cubic splines E02BAF
  with polynomials,
    arbitrary data points E02ADF
    with constraints E02AGF
Least-squares surface fit with bicubic splines E02DAF
  fit with general linear function, E02GAF
Sorting,
  2-D data into panels E02ZAF
E02AKF Evaluation of fitted polynomial in one variable, from Chebyshev series form

E02BAF Least-squares curve cubic spline fit (including interpolation)

E02BBF Evaluation of fitted cubic spline, function only

E02BCF Evaluation of fitted cubic spline, function and derivatives

E02BDF Evaluation of fitted cubic spline, definite integral

E02BEF Least-squares cubic spline curve fit, automatic knot placement

E02DAF Least-squares surface fit, bicubic splines

E02DCF Least-squares surface fit by bicubic splines with automatic knot placement, data on rectangular grid

E02DDF Least-squares surface fit by bicubic splines with automatic knot placement, scattered data

E02DEF Evaluation of a fitted bicubic spline at a vector of points

E02DFF Evaluation of a fitted bicubic spline at a mesh of points

E02GAF L₁-approximation by general linear function

E02ZAF Sort 2-D data into panels for fitting bicubic splines

E02ADF computes weighted least-squares polynomial approximations to an arbitrary set of data points.
2. Specification

SUBROUTINE E02ADF (M, KPLUS1, NROWS, X, Y, W, WORK1, IFAIL)
   PARAMETER (M, KPLUS1, NROWS, IFAIL)
   DOUBLE PRECISION X(M), Y(M), W(M), WORK1(3*M), WORK2(2*KPLUS1), A(NROWS,KPLUS1), S(KPLUS1)

3. Description

This routine determines least-squares polynomial approximations of degrees 0,1,...,k to the set of data points \((x_r, y_r)\) with weights \(w_r\), for \(r=1,2,...,m\).

The approximation of degree \(i\) has the property that it minimizes (sigma) the sum of squares of the weighted residuals \((\epsilon_r)\),

\[
(\epsilon_r) = w_r (y_r - f_r)
\]

and \(f_r\) is the value of the polynomial of degree \(i\) at the \(r\)th data point.

Each polynomial is represented in Chebyshev-series form with normalised argument \(x\). This argument lies in the range \(-1\) to \(+1\) and is related to the original variable \(x\) by the linear transformation

\[
\frac{2x - x_{\text{max}} - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} = \frac{2x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}.
\]

Here \(x_{\text{max}}\) and \(x_{\text{min}}\) are respectively the largest and smallest values of \(x\). The polynomial approximation of degree \(i\) is represented as

\[
1 -a_0 T(x) + a_1 T(x)^2 + a_2 T(x)^3 + \cdots + a_i T(x)^i,
\]
where \( T(x) \) is the Chebyshev polynomial of the first kind of degree \( j \) with argument \( x \).

For \( i=0,1,...,k \), the routine produces the values of \( a_{i+1,j+1} \) for \( j=0,1,...,i \), together with the value of the root mean square

\[
\frac{\text{residual}}{\sigma_i} = \frac{1}{i}\sqrt{\frac{\sum|y-a(x)|^2}{m-i-1}}
\]

In the case \( m=i+1 \) the routine sets the value of \( s \) to zero.

The method employed is due to Forsythe [4] and is based upon the generation of a set of polynomials orthogonal with respect to summation over the normalised data set. The extensions due to Clenshaw [1] to represent these polynomials as well as the approximating polynomials in their Chebyshev-series forms are incorporated. The modifications suggested by Reinsch and Gentleman (see [5]) to the method originally employed by Clenshaw for evaluating the orthogonal polynomials from their Chebyshev-series representations are used to give greater numerical stability.

For further details of the algorithm and its use see Cox [2] and [3].

Subsequent evaluation of the Chebyshev-series representations of the polynomial approximations should be carried out using E02AEF.

4. References


5. Parameters

1: M -- INTEGER
Input
On entry: the number m of data points. Constraint: M \geq MDIST \geq 2, where MDIST is the number of distinct x values in the data.

2: KPLUS1 -- INTEGER
Input
On entry: k+1, where k is the maximum degree required. Constraint: 0 < KPLUS1 \leq MDIST, where MDIST is the number of distinct x values in the data.

3: NROWS -- INTEGER
Input
On entry: the first dimension of the array A as declared in the (sub)program from which E02ADF is called. Constraint: NROWS \geq KPLUS1.

4: X(M) -- DOUBLE PRECISION array
Input
On entry: the values x of the independent variable, for \( r = 1, 2, \ldots, m \). Constraint: the values must be supplied in non-decreasing order with \( X(1) < X(M) \).

5: Y(M) -- DOUBLE PRECISION array
Input
On entry: the values y of the dependent variable, for \( r = 1, 2, \ldots, m \).

6: W(M) -- DOUBLE PRECISION array
Input
On entry: the set of weights, \( w \), for \( r = 1, 2, \ldots, m \). For advice on the choice of weights, see Section 2.1.2 of the Chapter Introduction. Constraint: \( W(r) > 0.0 \), for \( r = 1, 2, \ldots, m \).

7: WORK1(3*M) -- DOUBLE PRECISION array
Workspace
8: WORK2(2*KPLUS1) -- DOUBLE PRECISION array Workspace

9: A(NROWS,KPLUS1) -- DOUBLE PRECISION array Output

On exit: the coefficients of $T(x)$ in the approximating
\[ j \]
polynomial of degree $i$. $A(i+1,j+1)$ contains the coefficient
\[ a_{i+1,j+1} \]
for $i=0,1,...,k$; $j=0,1,...,i$.

10: S(KPLUS1) -- DOUBLE PRECISION array Output

On exit: $S(i+1)$ contains the root mean square residual $s_i$
for $i=0,1,...,k$, as described in Section 3. For the
interpretation of the values of the $s_i$ and their use in
selecting an appropriate degree, see Section 3.1 of the
Chapter Introduction.

11: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The weights are not all strictly positive.

IFAIL= 2
The values of $X(r)$, for $r=1,2,...,M$ are not in non-decreasing order.

IFAIL= 3
All $X(r)$ have the same value: thus the normalisation of $X$ is not possible.

IFAIL= 4
On entry $KPLUS1 < 1$ (so the maximum degree required is negative)

or $KPLUS1 > MDIST$, where $MDIST$ is the number of
distinct $x$ values in the data (so there cannot be a
unique solution for degree $k=KPLUS1-1$).
IFAIL= 5
    NRWNG < KPLUS1.

7. Accuracy
No error analysis for the method has been published. Practical experience with the method, however, is generally extremely satisfactory.

8. Further Comments
The time taken by the routine is approximately proportional to m(k+1)(k+11).

The approximating polynomials may exhibit undesirable oscillations (particularly near the ends of the range) if the maximum degree k exceeds a critical value which depends on the number of data points m and their relative positions. As a rough guide, for equally-spaced data, this critical value is about 2√m. For further details see Hayes [6] page 60.

9. Example
Determine weighted least-squares polynomial approximations of degrees 0, 1, 2 and 3 to a set of 11 prescribed data points. For the approximation of degree 3, tabulate the data and the corresponding values of the approximating polynomial, together with the residual errors, and also the values of the approximating polynomial at points half-way between each pair of adjacent data points.

The example program supplied is written in a general form that will enable polynomial approximations of degrees 0, 1, ..., k to be obtained to m data points, with arbitrary positive weights, and the approximation of degree k to be tabulated. E02AEF is used to evaluate the approximating polynomial. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AEF evaluates a polynomial from its Chebyshev-series representation.

2. Specification

```fortran
SUBROUTINE E02AEF (NPLUS1, A, XCAP, P, IFAIL)
INTEGER NPLUS1, IFAIL
DOUBLE PRECISION A(NPLUS1), XCAP, P
```

3. Description

This routine evaluates the polynomial

\[
1 - a_T(x) + a_{T_1}(x) + a_{T_2}(x) + \ldots + a_{T_{n+1}}(x)
\]

for any value of x satisfying \(-1 \leq x \leq 1\). Here \(T_j(x)\) denotes the Chebyshev polynomial of the first kind of degree \(j\) with argument \(x\). The value of \(n\) is prescribed by the user.

In practice, the variable \(x\) will usually have been obtained from an original variable \(x\), where \(\min x \leq x \leq \max x\) and

\[
((x-x_{\min})-(x-x_{\max}))
\]

\[
\min x_{\max}
\]

\[
x = \frac{(x-x_{\min})}{(x_{\max}-x_{\min})}
\]

Note that this form of the transformation should be used computationally rather than the mathematical equivalent

\[
(2x-x_{\min}-x_{\max})
\]

\[
\min x_{\max}
\]
since the former guarantees that the computed value of \( x \) differs from its true value by at most 4(\( \varepsilon \)), where (\( \varepsilon \)) is the machine precision, whereas the latter has no such guarantee.

The method employed is based upon the three-term recurrence relation due to Clenshaw [1], with modifications to give greater numerical stability due to Reinsch and Gentleman (see [4]).

For further details of the algorithm and its use see Cox [2] and [3].

4. References


5. Parameters

1: NPLUS1 -- INTEGER Input
On entry: the number \( n+1 \) of terms in the series (i.e., one greater than the degree of the polynomial). Constraint: NPLUS1 \( \geq 1 \).

2: A(NPLUS1) -- DOUBLE PRECISION array Input
On entry: \( A(i) \) must be set to the value of the \( i \)th coefficient in the series, for \( i = 1, 2, \ldots, n+1 \).

3: XCAP -- DOUBLE PRECISION Input
On entry: \( x \), the argument at which the polynomial is to be evaluated. It should lie in the range \(-1\) to \(+1\), but a value
just outside this range is permitted (see Section 6) to allow for possible rounding errors committed in the

transformation from \( x \) to \( x \) discussed in Section 3. Provided the recommended form of the transformation is used, a successful exit is thus assured whenever the value of \( x \) lies in the range \( x_{\text{min}} \) to \( x_{\text{max}} \).

4: \( P \) -- DOUBLE PRECISION Output
   On exit: the value of the polynomial.

5: \( IFAIL \) -- INTEGER Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\( IFAIL = 1 \)
   \( \text{ABS}(X\text{CAP}) > 1.0 + 4(\epsilon) \), where \( (\epsilon) \) is the machine precision. In this case the value of \( P \) is set arbitrarily to zero.

\( IFAIL = 2 \)
   On entry \( NPLUS1 < 1 \).

7. Accuracy

The rounding errors committed are such that the computed value of the polynomial is exact for a slightly perturbed set of coefficients \( a_i + (\delta)a_i \). The ratio of the sum of the absolute values of the \( (\delta)a_i \) to the sum of the absolute values of the \( a_i \) is less than a small multiple of \((n+1)\) times machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to \( n+1 \).
It is expected that a common use of E02AEF will be the evaluation of the polynomial approximations produced by E02ADF and E02AFF(*)

9. Example

Evaluate at 11 equally-spaced points in the interval -1<=x<=1 the polynomial of degree 4 with Chebyshev coefficients, 2.0, 0.5, 0.25, 0.125, 0.0625.

The example program is written in a general form that will enable a polynomial of degree n in its Chebyshev-series form to be evaluated at m equally-spaced points in the interval -1<=x<=1. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

SUBROUTINE E02AGF (M, KPLUS1, NROWS, XMIN, XMAX, X, Y, W, MF, XF, YF, LYF, IP, A, S, NP1, WRK, LWRK, IWRK, LIWRK, IFAIL)

E02AGF computes constrained weighted least-squares polynomial approximations in Chebyshev-series form to an arbitrary set of data points. The values of the approximations and any number of their derivatives can be specified at selected points.

1. Purpose

2. Specification
3. Description

This routine determines least-squares polynomial approximations of degrees up to \( k \) to the set of data points \((x_r, y_r)\) with weights \( w_r \), for \( r = 1, 2, \ldots, m \). The value of \( k \), the maximum degree required, is prescribed by the user. At each of the values \( x_F^r \), for \( r = 1, 2, \ldots, M_F \), of the independent variable \( x \), the approximations and their derivatives up to order \( p \) are constrained to have one of the user-specified values \( Y_F^s \), for \( s = 1, 2, \ldots, n \), where \( n = M_F + p \).

The approximation of degree \( i \) has the property that, subject to the imposed constraints, it minimizes \((\Sigma) \), the sum of the squares of the weighted residuals \((\epsilon)\) for \( r = 1, 2, \ldots, m \) where

\[
(\epsilon) = w_r (y_r - f_i^r(x)^r)
\]

and \( f_i^r(x) \) is the value of the polynomial approximation of degree \( i \) at the \( r \)th data point.

Each polynomial is represented in Chebyshev-series form with normalised argument \( x \). This argument lies in the range \(-1\) to \(+1\) and is related to the original variable \( x \) by the linear transformation

\[
2x - (x_{\text{max}} + x_{\text{min}})
\]

\[
x = \frac{x_{\text{max}} - x_{\text{min}}}{x - x_{\text{min}}}
\]

where \( x_{\text{min}} \) and \( x_{\text{max}} \), specified by the user, are respectively the lower and upper end-points of the interval of \( x \) over which the polynomials are to be defined.
The polynomial approximation of degree $i$ can be written as

$$1 - a_0 + a_1 T(x) + ... + a_i T(x) + ... + a_{2i} T(x)$$

where $T(x)$ is the Chebyshev polynomial of the first kind of degree $j$ with argument $x$. For $i=n,n+1,...,k$, the routine produces the values of the coefficients $a_{ij}$, for $j=0,1,...,i$, together with the value of the root mean square residual, $S_i$, defined as

$$S_i = \sqrt{\frac{1}{m'} \sum_{r=1}^{m'} (y_r - (mu)(x_r))^2}$$

where $m'$ is the number of data points with non-zero weight.

Values of the approximations may subsequently be computed using E02AEF or E02AKF.

First E02AGF determines a polynomial $(mu)(x)$, of degree $n-1$, which satisfies the given constraints, and a polynomial $(nu)(x)$, of degree $n$, which has value (or derivative) zero wherever a constrained value (or derivative) is specified. It then fits $y - (mu)(x)$, for $r=1,2,...,m$ with polynomials of the required degree in $x$ each with factor $(nu)(x)$. Finally the coefficients of $(mu)(x)$ are added to the coefficients of these fits to give the coefficients of the constrained polynomial approximations to the data points $(x_r, y_r)$, for $r=1,2,...,m$. The method employed is given in Hayes [3]: it is an extension of Forsythe’s orthogonal polynomials method [2] as modified by Clenshaw [1].
4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number m of data points to be fitted.
   Constraint: M >= 1.

2: KPLUS1 -- INTEGER Input
   On entry: k+1, where k is the maximum degree required.
   Constraint: n+1<=KPLUS1<=m''+n, where n is the total number of constraints and m'' is the number of data points with non-zero weights and distinct abscissae which do not coincide with any of the XF(r).

3: NROWS -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which E02AGF is called.
   Constraint: NROWS >= KPLUS1.

4: XMIN -- DOUBLE PRECISION Input

5: XMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper end-points, respectively, of the interval [x_min,x_max]. Unless there are specific reasons to the contrary, it is recommended that XMIN and XMAX be set respectively to the lowest and highest value among the x_r and XF(r). This avoids the danger of extrapolation provided there is a constraint point or data point with non-zero weight at each end-point. Constraint: XMAX > XMIN.

6: X(M) -- DOUBLE PRECISION array Input
   On entry: the value x_r of the independent variable at the r_th data point, for r=1,2,...,m. Constraint: the X(r) must be
in non-decreasing order and satisfy $X_{\text{MIN}} \leq X(r) \leq X_{\text{MAX}}$. 

7: Y(M) -- DOUBLE PRECISION array
On entry: $Y(r)$ must contain $y$, the value of the dependent variable at the $r$th data point, for $r=1,2,\ldots,m$. 

8: W(M) -- DOUBLE PRECISION array
On entry: the weights $w$ to be applied to the data points $x$, for $r=1,2,\ldots,m$. For advice on the choice of weights see the Chapter Introduction. Negative weights are treated as positive. A zero weight causes the corresponding data point to be ignored. Zero weight should be given to any data point whose $x$ and $y$ values both coincide with those of a constraint (otherwise the denominators involved in the root-mean-square residuals $s$ will be slightly in error).

9: MF -- INTEGER
On entry: the number of values of the independent variable at which a constraint is specified. Constraint: $MF \geq 1$. 

10: XF(MF) -- DOUBLE PRECISION array
On entry: the $r$th value of the independent variable at which a constraint is specified, for $r = 1,2,\ldots,\text{MF}$. Constraint: these values need not be ordered but must be distinct and satisfy $X_{\text{MIN}} \leq XF(r) \leq X_{\text{MAX}}$. 

11: YF(LYF) -- DOUBLE PRECISION array
On entry: the values which the approximating polynomials and their derivatives are required to take at the points specified in XF. For each value of $XF(r)$, YF contains in successive elements the required value of the approximation, its first derivative, second derivative, $\ldots$, $p$ th derivative, for $r = 1,2,\ldots,\text{MF}$. Thus the value which the $k$th derivative of each approximation ($k=0$ referring to the approximation itself) is required to take at the point $XF(r)$ must be contained in $YF(s)$, where 

$$s=r+k+p+p+\ldots+p,$$

for $k=0,1,\ldots,p$ and $r = 1,2,\ldots,\text{MF}$. The derivatives are with respect to the user's variable $x$. 

12: LYF -- INTEGER
On entry: the dimension of the array YF as declared in the
(sub)program from which E02AGF is called.
Constraint: LYF>=n, where n=MF+p +p +...+p .

1 2  MF

13: IP(MF) -- INTEGER array
Input
On entry: IP(r) must contain p , the order of the highest-
order derivative specified at XF(r), for r = 1,2,...,MF.
p =0 implies that the value of the approximation at XF(r) is
specified, but not that of any derivative. Constraint: IP(r)
>= 0, for r=1,2,...,MF.

14: A(NROWS,KPLUS1) -- DOUBLE PRECISION array
Output
On exit: A(i+1,j+1) contains the coefficient a in the
ij approximating polynomial of degree i, for i=n,n+1,...,k;
j=0,1,...,i.

15: S(KPLUS1) -- DOUBLE PRECISION array
Output
On exit: S(i+1) contains s , for i=n,n+1,...,k, the root-
i mean-square residual corresponding to the approximating
polynomial of degree i. In the case where the number of data
points with non-zero weight is equal to k+1-n, s is
indeterminate: the routine sets it to zero. For the
interpretation of the values of s and their use in
selecting an appropriate degree, see Section 3.1 of the
Chapter Introduction.

16: NP1 -- INTEGER
Output
On exit: n+1, where n is the total number of constraint
conditions imposed: n=MF+p +p +...+p .
1 2  MF

17: WRK(LWRK) -- DOUBLE PRECISION array
Output
On exit: WRK contains weighted residuals of the highest
degree of fit determined (k). The residual at x is in
element 2(n+1)+3(m+k+1)+r, for r=1,2,...,m. The rest of the
array is used as workspace.

18: LWRK -- INTEGER
Input
On entry:
the dimension of the array WRK as declared in the
(sub)program from which E02AGF is called.
Constraint: LWRK>=max(4*M+3*KPLUS1,8*n+5*IPMAX+MF+10)+2*n+2
, where IPMAX = max(IP(R)).
19: IWRK(LIWRK) -- INTEGER array  
   Workspace

20: LIWRK -- INTEGER  
   Input  
   On entry:  
   the dimension of the array IWRK as declared in the  
   (sub)program from which E02AGF is called.  
   Constraint: LIWRK>=2*MF+2.

21: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1  
On entry M < 1,  
   or    KPLUS1 < n + 1,  
   or    NROWS < KPLUS1,  
   or    MF < 1,  
   or    LWF < n,  
   or    LWRK is too small (see Section 5),  
   or    LIWRK<2*MF+2.  
(Here n is the total number of constraint conditions.)

IFAIL= 2  
IP(r) < 0 for some r = 1,2,...,MF.

IFAIL= 3  
XMIN >= XMAX, or XF(r) is not in the interval XMIN to XMAX  
for some r = 1,2,...,MF, or the XF(r) are not distinct.

IFAIL= 4  
X(r) is not in the interval XMIN to XMAX for some  
r=1,2,...,M.

IFAIL= 5  
X(r) < X(r-1) for some r=2,3,...,M.
IFAIL= 6
  KPLUS1>m''+n, where m'' is the number of data points with non-zero weight and distinct abscissae which do not coincide with any XF(r). Thus there is no unique solution.

IFAIL= 7
  The polynomials (mu)(x) and/or (nu)(x) cannot be determined. The problem supplied is too ill-conditioned. This may occur when the constraint points are very close together, or large in number, or when an attempt is made to constrain high-order derivatives.

7. Accuracy

No complete error analysis exists for either the interpolating algorithm or the approximating algorithm. However, considerable experience with the approximating algorithm shows that it is generally extremely satisfactory. Also the moderate number of constraints, of low order, which are typical of data fitting applications, are unlikely to cause difficulty with the interpolating routine.

8. Further Comments

The time taken by the routine to form the interpolating polynomial is approximately proportional to n^3, and that to form the approximating polynomials is very approximately proportional to m(k+1)(k+1-n).

To carry out a least-squares polynomial fit without constraints, use E02ADF. To carry out polynomial interpolation only, use E01AEF(*).

9. Example

The example program reads data in the following order, using the notation of the parameter list above:

    MF
    IP(i), XF(i), Y-value and derivative values (if any) at XF(i), for i= 1,2,...,MF

    M
    X(i), Y(i), W(i), for i=1,2,...,M

    k, XMIN, XMAX
The output is:

the root-mean-square residual for each degree from n to k;
the Chebyshev coefficients for the fit of degree k;
the data points, and the fitted values and residuals for
the fit of degree k.

The program is written in a generalized form which will read any
number of data sets.

The data set supplied specifies 5 data points in the interval [0.0, 0.4.0] with unit weights, to which are to be fitted polynomials, p, of degrees up to 4, subject to the 3 constraints:

\[ p(0.0) = 1.0, \quad p'(0.0) = -2.0, \quad p(4.0) = 9.0. \]

The example program is not reproduced here. The source code for
each example program is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting

E02AHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02AHF determines the coefficients in the Chebyshev-series
representation of the derivative of a polynomial given in
Chebyshev-series form.

2. Specification

```
SUBROUTINE E02AHF (NP1, XMIN, XMAX, A, IA1, LA, PATM1, ADIF, IADIF1, LADIF, IFAIL)
INTEGER NP1, IA1, LA, IADIF1, LADIF, IFAIL
DOUBLE PRECISION XMIN, XMAX, A(LA), PATM1, ADIF(LADIF)
```

3. Description

This routine forms the polynomial which is the derivative of a
given polynomial. Both the original polynomial and its derivative
are represented in Chebyshev-series form. Given the coefficients
a, for \(i=0,1,...,n\), of a polynomial \(p(x)\) of degree \(n\), where
\[
p(x) = -a + a_1 T_1(x) + ... + a_n T_n(x)
\]
the routine returns the coefficients \(a_i\), for \(i=0,1,...,n-1\), of
the polynomial \(q(x)\) of degree \(n-1\), where
\[
\frac{dp(x)}{dx} = -a + a_1 T_1(x) + ... + a_{n-1} T_{n-1}(x).
\]
Here \(T_j(x)\) denotes the Chebyshev polynomial of the first kind of
degree \(j\) with argument \(x\). It is assumed that the normalised
variable \(x\) in the interval \([-1,+1]\) was obtained from the user's
original variable \(x\) in the interval \([x_{\text{min}},x_{\text{max}}]\) by the linear
transformation
\[
x = \frac{2x - (x_{\text{max}} + x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}}
\]
and that the user requires the derivative to be with respect to
the variable \(x\). If the derivative with respect to \(x\) is required,
set \(x_{\text{max}} = 1\) and \(x_{\text{min}} = -1\).
Values of the derivative can subsequently be computed, from the
coefficients obtained, by using E02AKF.
The method employed is that of [1] modified to obtain the
derivative with respect to $x$. Initially setting $a = a = 0$, the
$n+1$ $n$
routine forms successively

\[ a = a + \frac{-2ia}{i(i+1)} \cdot x^i \quad i = n, n-1, \ldots, 1. \]

\[ i-1 \quad i+1 \quad x^i \quad x_{\text{min}} \quad x_{\text{max}} \]

4. References

HMSO.

5. Parameters

1: NP1 -- INTEGER Input
On entry: $n+1$, where $n$ is the degree of the given
polynomial $p(x)$. Thus NP1 is the number of coefficients in
this polynomial. Constraint: NP1 $\geq 1$.

2: XMIN -- DOUBLE PRECISION Input
3: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points respectively of
the interval $[x_{\text{min}}, x_{\text{max}}]$. The Chebyshev-series
representation is in terms of the normalised variable $x$, where

\[ 2x - (x_{\text{max}} + x_{\text{min}}) \]
\[ x = \frac{2x - (x_{\text{max}} + x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}} \]

Constraint: XMAX > XMIN.

4: A(LA) -- DOUBLE PRECISION array Input
On entry: the Chebyshev coefficients of the polynomial $p(x)$.
Specifically, element $1 + i*IA1$ of A must contain the
coefficient $a_i$ for $i = 0, 1, \ldots, n$. Only these $n+1$ elements
will be accessed.

Unchanged on exit, but see ADIF, below.

5: IA1 -- INTEGER Input
On entry: the index increment of A. Most frequently the
Chebyshev coefficients are stored in adjacent elements of A, and IA1 must be set to 1. However, if, for example, they are stored in A(1), A(4), A(7), ..., then the value of IA1 must be 3. See also Section 8. Constraint: IA1 >= 1.

6: LA -- INTEGER  
Input
On entry:  
the dimension of the array A as declared in the (sub)program from which E02AHF is called.  
Constraint: LA>=1+(NP1-1)*IA1.

7: PATM1 -- DOUBLE PRECISION  
Output
On exit: the value of p(x ). If this value is passed to  
min  
the integration routine E02AJF with the coefficients of q(x)  
, then the original polynomial p(x) is recovered, including  
its constant coefficient.

8: ADIF(LADIF) -- DOUBLE PRECISION array  
Output
On exit: the Chebyshev coefficients of the derived polynomial q(x). (The differentiation is with respect to the variable x). Specifically, element 1+i*IADIF1 of ADIF contains the coefficient a , i=0,1,...,n-1. Additionally  
i  
element 1+n*IADIF1 is set to zero. A call of the routine may have the array name ADIF the same as A, provided that note is taken of the order in which elements are overwritten, when choosing the starting elements and increments IA1 and IADIF1: i.e., the coefficients a ,a ,...,a must be intact 0 1 i-1  
i

after coefficient a is stored. In particular, it is  
i
possible to overwrite the a completely by having IA1 =  
i  
IADIF1, and the actual arrays for A and ADIF identical.

9: IADIF1 -- INTEGER  
Input
On entry: the index increment of ADIF. Most frequently the Chebyshev coefficients are required in adjacent elements of ADIF, and IADIF1 must be set to 1. However, if, for example, they are to be stored in ADIF(1),ADIF(4),ADIF(7),..., then the value of IADIF1 must be 3. See Section 8. Constraint: IADIF1 >= 1.

10: LADIF -- INTEGER  
Input
On entry:
the dimension of the array ADIF as declared in the
(sub)program from which E02AHF is called.
Constraint: LADIF\geq 1+(NP1-1)*IADIF1.

11: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry NP1 < 1,
or XMAX <= XMIN,
or IA1 < 1,
or LAX=(NP1-1)*IA1,
or IADIF1 < 1,
or LADIF<=(NP1-1)*IADIF1.

7. Accuracy

There is always a loss of precision in numerical differentiation,
in this case associated with the multiplication by 2i in the
formula quoted in Section 3.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

The increments IA1, IADIF1 are included as parameters to give a
degree of flexibility which, for example, allows a polynomial in
two variables to be differentiated with respect to either
variable without rearranging the coefficients.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the 1st and 2nd derivatives of this polynomial at 4
equally spaced points over the interval. (For the purposes of this example, XMIN, XMAX and the Chebyshev coefficients are simply supplied in DATA statements. Normally a program would first read in or generate data and compute the fitted polynomial.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

E02 -- Curve and Surface Fitting  
E02AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AJF determines the coefficients in the Chebyshev-series representation of the indefinite integral of a polynomial given in Chebyshev-series form.

2. Specification

```fortran
SUBROUTINE E02AJF (NP1, XMIN, XMAX, A, IA1, LA, QATM1, 
                    AINT, IAINT1, LAINT, IFAIL)
    INTEGER NP1, IA1, LA, IAINT1, LAINT, IFAIL
    DOUBLE PRECISION XMIN, XMAX, A(LA), QATM1, AINT(LAINT)
```

3. Description

This routine forms the polynomial which is the indefinite integral of a given polynomial. Both the original polynomial and its integral are represented in Chebyshev-series form. If supplied with the coefficients $a_i$ for $i=0,1,...,n$, of a polynomial $p(x)$ of degree $n$, where

$$p(x) = -a_0 T_0(x) + a_1 T_1(x) + ... + a_n T_n(x),$$

the routine returns the coefficients $a'_i$, for $i=0,1,...,n+1$, of the polynomial $q(x)$ of degree $n+1$, where

$$q(x) = \int p(x) \, dx.$$
\[ q(x) = -a' + a' T_1(x) + \ldots + a' T_{n+1}(x), \]

and

\[ / \]

\[ q(x) = |p(x)dx. \]

Here \( T_j(x) \) denotes the Chebyshev polynomial of the first kind of degree \( j \) with argument \( x \). It is assumed that the normalised variable \( x \) in the interval \([-1, 1]\) was obtained from the user’s original variable \( x \) in the interval \([a, b]\) by the linear transformation

\[
\frac{2x-(a+b)}{b-a}
\]

and that the user requires the integral to be with respect to the variable \( x \). If the integral with respect to \( x \) is required, set \( x = 1 \) and \( x = -1 \).

Values of the integral can subsequently be computed, from the coefficients obtained, by using E02AKF.

The method employed is that of Chebyshev-series [1] modified for integrating with respect to \( x \). Initially taking \( a = a = 0 \), the routine forms successively

\[
a' = \frac{a - a' x - x}{2i 2i}, \quad i = n+1, n+2
\]

The constant coefficient \( a' \) is chosen so that \( q(x) \) is equal to \( a_0 \).
specified value, QATM1, at the lower end-point of the interval on which it is defined, i.e., x=-1, which corresponds to x=x_{min}.

4. References


5. Parameters

1: NP1 -- INTEGER Input
   On entry: n+1, where n is the degree of the given polynomial p(x). Thus NP1 is the number of coefficients in this polynomial. Constraint: NP1 \geq 1.

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper end-points respectively of the interval [x_{min},x_{max}]. The Chebyshev-series representation is in terms of the normalised variable x, where
   \[
   2x - (x_{max} + x_{min}) \\
   x = \frac{2x_{min} - x}{x_{max} - x_{min}}.
   \]
   Constraint: XMAX > XMIN.

4: A(LA) -- DOUBLE PRECISION array Input
   On entry: the Chebyshev coefficients of the polynomial p(x). Specifically, element 1+i*IA1 of A must contain the coefficient a_i, for i=0,1,...,n. Only these n+1 elements will be accessed.

   Unchanged on exit, but see AINT, below.

5: IA1 -- INTEGER Input
   On entry: the index increment of A. Most frequently the Chebyshev coefficients are stored in adjacent elements of A, and IA1 must be set to 1. However, if for example, they are stored in A(1),A(4),A(7),..., then the value of IA1 must be
3. See also Section 8. Constraint: IA1 >= 1.

6: LA -- INTEGER                   Input
    On entry:
    the dimension of the array A as declared in the (sub)program
    from which E02AJF is called.
    Constraint: LA>=1+(NP1-1)*IA1.

7: QATM1 -- DOUBLE PRECISION       Input
    On entry: the value that the integrated polynomial is
    required to have at the lower end-point of its interval of
    definition, i.e., at x=-1 which corresponds to x=x_min. Thus,
    QATM1 is a constant of integration and will normally be set
    to zero by the user.

8: AINT(LAINT) -- DOUBLE PRECISION array Output
    On exit: the Chebyshev coefficients of the integral q(x).
    (The integration is with respect to the variable x, and the
    constant coefficient is chosen so that q(x_min) equals QATM1)
    Specifically, element 1+i*IAINT1 of AINT contains the
    coefficient a'_i, for i=0,1,...,n+1. A call of the routine
    may have the array name AINT the same as A, provided that
    note is taken of the order in which elements are overwritten
    when choosing starting elements and increments IA1 and
    IAINTE: i.e., the coefficients, a_0,a_1,...,a_i must be
    intact after coefficient a'_i is stored. In particular it is
    possible to overwrite the a_i entirely by having IA1 =
    IAINTE, and the actual array for A and AINT identical.

9: IAINTE -- INTEGER                Input
    On entry: the index increment of AINT. Most frequently the
    Chebyshev coefficients are required in adjacent elements of
    AINT, and IAINTE must be set to 1. However, if, for example,
    they are to be stored in AINT(1), AINT(4), AINT(7),..., then
    the value of IAINTE must be 3. See also Section 8.
    Constraint: IAINTE >= 1.

10: LAINT -- INTEGER                 Input
    On entry:
    the dimension of the array AINT as declared in the
    (sub)program from which E02AJF is called.
    Constraint: LAINT>=1+NP1*IAINT.
11: IFAIL -- INTEGER
   Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry NP1 < 1,
or XMAX <= XMIN,
or IA1 < 1,
or LA<=(NP1-1)*IA1,
or IAINT1 < 1,
or LAINT<=NP1*IAINT1.

7. Accuracy

In general there is a gain in precision in numerical integration,
in this case associated with the division by 2i in the formula
quoted in Section 3.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

The increments IA1, IAINT1 are included as parameters to give a
degree of flexibility which, for example, allows a polynomial in
two variables to be integrated with respect to either variable
without rearranging the coefficients.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the integral of the polynomial from 0.0 to 2.0. (For
the purpose of this example, XMIN, XMAX and the Chebyshev
coefficients are simply supplied in DATA statements. Normally a
program would read in or generate data and compute the fitted
polynomial).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting
E02AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AKF evaluates a polynomial from its Chebyshev-series representation, allowing an arbitrary index increment for accessing the array of coefficients.

2. Specification

SUBROUTINE E02AKF (NP1, XMIN, XMAX, A, IA1, LA, X, RESULT, IFAIL)
INTEGER NP1, IA1, LA, IFAIL
DOUBLE PRECISION XMIN, XMAX, A(LA), X, RESULT

3. Description

If supplied with the coefficients $a_i$, for $i=0,1,...,n$, of a polynomial $p(x)$ of degree $n$, where

\[
p(x) = -a_0 + a_1 T_1(x) + \cdots + a_n T_n(x),
\]

this routine returns the value of $p(x)$ at a user-specified value of the variable $x$. Here $T_j(x)$ denotes the Chebyshev polynomial of order $j$.
the first kind of degree $j$ with argument $x$. It is assumed that

the independent variable $x$ in the interval $[-1,+1]$ was obtained from the user’s original variable $x$ in the interval $[x_{\text{min}}, x_{\text{max}}]$ by the linear transformation

$$x = \frac{2x - (x_{\text{max}} + x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}}.$$ 

The coefficients $a_i$ may be supplied in the array $A$, with any increment between the indices of array elements which contain successive coefficients. This enables the routine to be used in surface fitting and other applications, in which the array might have two or more dimensions.

The method employed is based upon the three-term recurrence relation due to Clenshaw [1], with modifications due to Reinsch and Gentleman (see [4]). For further details of the algorithm and its use see Cox [2] and Cox and Hayes [3].

4. References


5. Parameters

1: NP1 -- INTEGER Input
   On entry: $n+1$, where $n$ is the degree of the given polynomial $p(x)$. Constraint: NP1 $\geq 1$. 

2: XMIN -- DOUBLE PRECISION  
   Input

3: XMAX -- DOUBLE PRECISION  
   Input  
   On entry: the lower and upper end-points respectively of the interval \([x_\min, x_\max]\). The Chebyshev-series representation is in terms of the normalised variable \(x\), where

\[
2x-(x +x )
\]

\[
\frac{x}{x -x}
\]

\[
\text{max} \quad \text{min}
\]

Constraint: XMIN < XMAX.

4: A(LA) -- DOUBLE PRECISION array  
   Input  
   On entry: the Chebyshev coefficients of the polynomial \(p(x)\). Specifically, element \(1+i*IA1\) must contain the coefficient \(a_i\), for \(i=0,1,...,n\). Only these \(n+1\) elements will be accessed.

5: IA1 -- INTEGER  
   Input  
   On entry: the index increment of \(A\). Most frequently, the Chebyshev coefficients are stored in adjacent elements of \(A\), and IA1 must be set to 1. However, if, for example, they are stored in \(A(1),A(4),A(7),...,\) then the value of IA1 must be 3. Constraint: IA1 >= 1.

6: LA -- INTEGER  
   Input  
   On entry: the dimension of the array \(A\) as declared in the (sub)program from which E02AKF is called. Constraint: LA >= (NP1-1)*IA1+1.

7: X -- DOUBLE PRECISION  
   Input  
   On entry: the argument \(x\) at which the polynomial is to be evaluated. Constraint: XMIN <= X <= XMAX.

8: RESULT -- DOUBLE PRECISION  
   Output  
   On exit: the value of the polynomial \(p(x)\).

9: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry NP1 < 1,
or IA1 < 1,
or LA<=(NP1-1)*IA1,
or XMIN >= XMAX.

IFAIL= 2
X does not satisfy the restriction XMIN <= X <= XMAX.

7. Accuracy

The rounding errors are such that the computed value of the polynomial is exact for a slightly perturbed set of coefficients \( a + (\delta)a \). The ratio of the sum of the absolute values of the \( \delta a \) to the sum of the absolute values of the \( a \) is less than a small multiple of \((n+1)\)\*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to n+1.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form to fit data over the interval \([-0.5,2.5]\). The example program evaluates the polynomial at 4 equally spaced points over the interval. (For the purposes of this example, XMIN, XMAX and the Chebyshev coefficients are supplied in DATA statements. Normally a program would first read in or generate data and compute the fitted polynomial.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
E02 -- Curve and Surface Fitting

E02BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BAF computes a weighted least-squares approximation to an arbitrary set of data points by a cubic spline with knots prescribed by the user. Cubic spline interpolation can also be carried out.

2. Specification

```
SUBROUTINE E02BAF (M, NCAP7, X, Y, W, LAMDA, WORK1, WORK2,
1                  C, SS, IFAIL)
INTEGER M, NCAP7, IFAIL
DOUBLE PRECISION X(M), Y(M), W(M), LAMDA(NCAP7), WORK1(M),
1                  WORK2(4*NCAP7), C(NCAP7), SS
```

3. Description

This routine determines a least-squares cubic spline approximation \( s(x) \) to the set of data points \((x_r, y_r)\) with weights \( w_r \), for \( r=1,2,\ldots,m \). The value of \( NCAP7 = n+7 \), where \( n \) is the number of intervals of the spline (one greater than the number of interior knots), and the values of the knots \( \lambda_5, \lambda_6, \ldots, \lambda_{n+3} \), interior to the data interval, are prescribed by the user.

\( s(x) \) has the property that it minimizes \( \theta \), the sum of squares of the weighted residuals \( \epsilon_r \), for \( r=1,2,\ldots,m \), where

\[
\epsilon_r = w_r (y_r - s(x_r)).
\]

The routine produces this minimizing value of \( \theta \) and the
coefficients $c_1, c_2, \ldots, c_q$, where $q=n+3$, in the B-spline
representation

\[ s(x) = \sum_{i=1}^{q} c_i N_i(x). \]

Here $N_i(x)$ denotes the normalised B-spline of degree 3 defined
upon the knots $(\lambda_{i-4}), (\lambda_{i-3}), \ldots, (\lambda_{i+4})$.

In order to define the full set of B-splines required, eight
additional knots $(\lambda_{n+4}), (\lambda_{n+5}), (\lambda_{n+6}), (\lambda_{n+7})$
and $(\lambda_{i-1}), (\lambda_{i}), (\lambda_{i+1}), (\lambda_{i+2})$ are inserted
automatically by the routine. The first four of these are set
equal to the smallest $x$ and the last four to the largest $x$.

The representation of $s(x)$ in terms of B-splines is the most
compact form possible in that only $n+3$ coefficients, in addition
to the $n+7$ knots, fully define $s(x)$.

The method employed involves forming and then computing the
least-squares solution of a set of $m$ linear equations in the
coefficients $c_i$ ($i=1, 2, \ldots, n+3$). The equations are formed using a
recurrence relation for B-splines that is unconditionally stable
(Cox [1], de Boor [5]), even for multiple (coincident) knots. The
least-squares solution is also obtained in a stable manner by
using orthogonal transformations, viz. a variant of Givens
rotations (Gentleman [6] and [7]). This requires only one
equation to be stored at a time. Full advantage is taken of the
structure of the equations, there being at most four non-zero
values of $N_i(x)$ for any value of $x$ and hence at most four
coefficients in each equation.

For further details of the algorithm and its use see Cox [2], [3]
and [4].
Subsequent evaluation of $s(x)$ from its B-spline representation may be carried out using E02BBF. If derivatives of $s(x)$ are also required, E02BCF may be used. E02BDF can be used to compute the definite integral of $s(x)$.

4. References


5. Parameters

1: M -- INTEGER
   Input
   On entry: the number $m$ of data points. Constraint: $M \geq MDIST \geq 4$, where MDIST is the number of distinct $x$ values in the data.

2: NCAP7 -- INTEGER
   Input
   On entry: $n+7$, where $n$ is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range $x$ to $x$)
over which the spline is defined. Constraint: \( 8 \leq NCAP7 \leq MDIST + 4 \), where MDIST is the number of distinct x values in the data.

3: \( X(M) \) -- DOUBLE PRECISION array Input
On entry: the values \( x \) of the independent variable \( r \) (abscissa), for \( r=1,2,...,m \). Constraint: \( x_1 \leq x_2 \leq ... \leq x_m \).

4: \( Y(M) \) -- DOUBLE PRECISION array Input
On entry: the values \( y \) of the dependent variable \( r \) (ordinate), for \( r=1,2,...,m \).

5: \( W(M) \) -- DOUBLE PRECISION array Input
On entry: the values \( w \) of the weights, for \( r=1,2,...,m \).
For advice on the choice of weights, see the Chapter Introduction. Constraint: \( W(r) > 0 \), for \( r=1,2,...,m \).

6: \( \text{LAMDA}(NCAP7) \) -- DOUBLE PRECISION array Input/Output
On entry: \( \text{LAMDA}(i) \) must be set to the \((i-4)\)th (interior) knot, \((\lambda_i)\), for \( i=5,6,...,n+3 \). Constraint: \( X(1) < \text{LAMDA}_i \leq \text{LAMDA}(6) \leq \text{LAMDA}(NCAP7-4) < X(M) \). On exit: the input values are unchanged, and \( \text{LAMDA}(i) \), for \( i = 1, 2, 3, 4, NCAP7-3, NCAP7-2, NCAP7-1, NCAP7 \) contains the additional (exterior) knots introduced by the routine. For advice on the choice of knots, see Section 3.3 of the Chapter Introduction.

7: \( \text{WORK1}(M) \) -- DOUBLE PRECISION array Workspace

8: \( \text{WORK2}(4\times NCAP7) \) -- DOUBLE PRECISION array Workspace

9: \( C(NCAP7) \) -- DOUBLE PRECISION array Output
On exit: the coefficient \( c_i \) of the B-spline \( N_i(x) \), for \( i=1,2,...,n+3 \). The remaining elements of the array are not used.

10: \( SS \) -- DOUBLE PRECISION Output
On exit: the residual sum of squares, \((\theta)\).

11: \( IFAIL \) -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The knots fail to satisfy the condition

\[ X(1) < LAMDA(5) <= LAMDA(6) <= \ldots <= LAMDA(NCAP7-4) < X(M). \]

Thus the knots are not in correct order or are not interior to the data interval.

IFAIL= 2
The weights are not all strictly positive.

IFAIL= 3
The values of \( X(r) \), for \( r = 1,2,\ldots,M \) are not in non-decreasing order.

IFAIL= 4
NCAP7 < 8 (so the number of interior knots is negative) or NCAP7 > MDIST + 4, where MDIST is the number of distinct x values in the data (so there cannot be a unique solution).

IFAIL= 5
The conditions specified by Schoenberg and Whitney [8] fail to hold for at least one subset of the distinct data abscissae. That is, there is no subset of NCAP7-4 strictly increasing values, \( X(R(1)), X(R(2)), \ldots, X(R(NCAP7-4)) \), among the abscissae such that

\[ X(R(1)) < LAMDA(1) < X(R(5)), \]
\[ X(R(2)) < LAMDA(2) < X(R(6)), \]
\[ \ldots \]
\[ X(R(NCAP7-8)) < LAMDA(NCAP7-8) < X(R(NCAP7-4)). \]

This means that there is no unique solution: there are regions containing too many knots compared with the number of data points.

7. Accuracy

The rounding errors committed are such that the computed
coefficients are exact for a slightly perturbed set of ordinates \( y + \delta y \). The ratio of the root-mean-square value for the
\[ r \]
(denoted \( r \)) to the root-mean-square value of the \( y \) can be expected
\[ r \]
to be less than a small multiple of \((\kappa) \cdot m \cdot \text{machine precision}\),
where \((\kappa)\) is a condition number for the problem. Values of
\((\kappa)\) for 20-30 practical data sets all proved to lie between
4.5 and 7.8 (see Cox [3]). (Note that for these data sets,
replacing the coincident end knots at the end-points \( x_1 \) and \( x_m \)
used in the routine by various choices of non-coincident exterior
knots gave values of \((\kappa)\) between 16 and 180. Again see Cox
[3] for further details.) In general we would not expect \((\kappa)\)
to be large unless the choice of knots results in near-violation
of the Schoenberg-Whitney conditions.

A cubic spline which adequately fits the data and is free from
spurious oscillations is more likely to be obtained if the knots
are chosen to be grouped more closely in regions where the
function (underlying the data) or its derivatives change more
rapidly than elsewhere.

8. Further Comments

The time taken by the routine is approximately \( C \cdot (2m+n+7) \)
seconds, where \( C \) is a machine-dependent constant.

Multiple knots are permitted as long as their multiplicity does
not exceed 4, i.e., the complete set of knots must satisfy

\[ \lambda_i < \lambda_{i+4} \]

for \( i = 1, 2, \ldots, n+3 \), (cf. Section 6). At a
\[ i \]
knot of multiplicity one (the usual case), \( s(x) \) and its first two
derivatives are continuous. At a knot of multiplicity two, \( s(x) \)
and its first derivative are continuous. At a knot of
multiplicity three, \( s(x) \) is continuous, and at a knot of
multiplicity four, \( s(x) \) is generally discontinuous.

The routine can be used efficiently for cubic spline

interpolation, i.e., if \( m = n+3 \). The abscissae must then of course
satisfy \( x_1 < x_2 < \ldots < x_m \). Recommended values for the knots in this
\[ 1 \quad 2 \quad m \]
case are $(\lambda_i) = x_i$, for $i=5,6,\ldots,n+3$.

9. Example

Determine a weighted least-squares cubic spline approximation with five intervals (four interior knots) to a set of 14 given data points. Tabulate the data and the corresponding values of the approximating spline, together with the residual errors, and also the values of the approximating spline at points half-way between each pair of adjacent data points.

The example program is written in a general form that will enable

a cubic spline approximation with $n$ intervals ($n-1$ interior knots) to be obtained to $m$ data points, with arbitrary positive weights, and the approximation to be tabulated. Note that E02BBF is used to evaluate the approximating spline. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E02BBF evaluates a cubic spline from its B-spline representation.

2. Specification

```fortran
SUBROUTINE E02BBF (NCAP7, LAMDA, C, X, S, IFAIL)
INTEGER NCAP7, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S
```

3. Description

This routine evaluates the cubic spline $s(x)$ at a prescribed argument $x$ from its augmented knot set $(\lambda_i)$, for $i=5,6,\ldots,n+3$. Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
i=1,2,...,n+7, (see E02BAF) and from the coefficients c, for
i
i=1,2,...,q in its B-spline representation

\[ s(x) = \sum_{i=1}^{q} c_i N_i(x) \]

Here q=n+3, where n is the number of intervals of the spline, and
\( N_i(x) \) denotes the normalised B-spline of degree 3 defined upon
i
the knots \( \lambda_1, \lambda_2, \ldots, \lambda_{n+4} \). The prescribed
i
i+1
i+4
argument x must satisfy \( \lambda_i \leq x \leq \lambda_{i+4} \).

It is assumed that \( \lambda_j \geq \lambda_{j-1} \), for \( j=2,3,\ldots,n+7 \), and
\( \lambda_j > \lambda_{j-1} \),
\[ n+4 \]

The method employed is that of evaluation by taking convex
combinations due to de Boor [4]. For further details of the
algorithm and its use see Cox [1] and [3].

It is expected that a common use of E02BBF will be the evaluation
of the cubic spline approximations produced by E02BAF. A
generalization of E02BBF which also forms the derivative of \( s(x) \)
is E02BCF. E02BCF takes about 50\% longer than E02BBF.

4. References


suite of algorithms for the non-specialist user. Report
5. Parameters

1: NCAP7 -- INTEGER  
   Input
   On entry: \( n+7 \), where \( n \) is the number of intervals (one greater than the number of interior knots, i.e., the knots strictly within the range \((\lambda_1)\) to \((\lambda_n)\) ) over which the spline is defined. Constraint: \( NCAP7 \geq 8 \).

2: LAMDA(NCAP7) -- DOUBLE PRECISION array  
   Input
   On entry: \( \text{LAMDA}(j) \) must be set to the value of the \( j \)th member of the complete set of knots, \((\lambda_j)\) for \( j=1,2,...,n+7 \). Constraint: the \( \text{LAMDA}(j) \) must be in non-decreasing order with \( \text{LAMDA}(NCAP7-3) > \text{LAMDA}(4) \).

3: C(NCAP7) -- DOUBLE PRECISION array  
   Input
   On entry: the coefficient \( c_i \) of the B-spline \( N_i(x) \), for \( i=1,2,...,n+3 \). The remaining elements of the array are not used.

4: X -- DOUBLE PRECISION  
   Input
   On entry: the argument \( x \) at which the cubic spline is to be evaluated. Constraint: \( \text{LAMDA}(4) \leq x \leq \text{LAMDA}(NCAP7-3) \).

5: S -- DOUBLE PRECISION  
   Output
   On exit: the value of the spline, \( s(x) \).

6: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
   The argument X does not satisfy LAMDA(4) <= X <= LAMDA(NCAP7-3).
   In this case the value of S is set arbitrarily to zero.

IFAIL= 2
   NCAP7 < 8, i.e., the number of interior knots is negative.

7. Accuracy

The computed value of s(x) has negligible error in most practical situations. Specifically, this value has an absolute error bounded in modulus by 18*c machine precision, where c is the largest in modulus of c, c, c, and c, and j is an integer such that (lambda) <=x<=(lambda). If c, c, c, c are all of the same sign, then the computed value of s(x) has a relative error not exceeding 20*machine precision in modulus. For further details see Cox [2].

8. Further Comments

The time taken by the routine is approximately C*(1+0.1*log(n+7)) seconds, where C is a machine-dependent constant.

Note: the routine does not test all the conditions on the knots given in the description of LAMDA in Section 5, since to do this would result in a computation time approximately linear in n+7 instead of log(n+7). All the conditions are tested in E02BAF, however.

9. Example

Evaluate at 9 equally-spaced points in the interval 1.0<=x<=9.0 the cubic spline with (augmented) knots 1.0, 1.0, 1.0, 1.0, 3.0, 6.0, 8.0, 9.0, 9.0, 9.0, 9.0 and normalised cubic B-spline coefficients 1.0, 2.0, 4.0, 7.0, 6.0, 4.0, 3.0.

The example program is written in a general form that will enable a cubic spline with n intervals, in its normalised cubic B-spline
form, to be evaluated at \( m \) equally-spaced points in the interval

\[ \text{LAMDA}(4) \leq x \leq \text{LAMDA}(n+4) \]. The program is self-starting in that any number of data sets may be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

```plaintext
E02 -- Curve and Surface Fitting E02BCF
E02BCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BCF evaluates a cubic spline and its first three derivatives from its B-spline representation.

2. Specification

```plaintext
SUBROUTINE E02BCF (NCAP7, LAMDA, C, X, LEFT, S, IFAIL)
INTEGER NCAP7, LEFT, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S(4)
```

3. Description

This routine evaluates the cubic spline \( s(x) \) and its first three derivatives at a prescribed argument \( x \). It is assumed that \( s(x) \) is represented in terms of its B-spline coefficients \( c_i \), for \( i=1,2,\ldots,n+3 \) and (augmented) ordered knot set \( \text{LAMDA} \), for \( i=1,2,\ldots,n+7 \), (see E02BAF), i.e.,

\[ q -- \]

\[ s(x) = \sum_{i=1}^{n+3} c_i N_i(x) \]

\[ -- i=1 \]
Here \( q = n + 3 \), \( n \) is the number of intervals of the spline and \( N(x) \)
denotes the normalised B-spline of degree 3 (order 4) defined
upon the knots \( (\lambda_i), (\lambda_{i+1}), \ldots, (\lambda_{i+4}) \). The
prescribed argument \( x \) must satisfy
\[
(\lambda_i) \leq x \leq (\lambda_{i+4})
\]
At a simple knot \( (\lambda_i) \) (i.e., one satisfying
\( (\lambda_{i-1}) < (\lambda_i) < (\lambda_{i+1}) \), the third derivative of the
spline is in general discontinuous. At a multiple knot (i.e., two
or more knots with the same value), lower derivatives, and even
the spline itself, may be discontinuous. Specifically, at a point
\( x = u \) where (exactly) \( r \) knots coincide (such a point is termed a
knot of multiplicity \( r \)), the values of the derivatives of order
\( 4 - j \), for \( j = 1, 2, \ldots, r \), are in general discontinuous. (Here
\( 1 \leq r \leq 4; r > 4 \) is not meaningful.) The user must specify whether the
value at such a point is required to be the left- or right-hand
derivative.

The method employed is based upon:

(i) carrying out a binary search for the knot interval
containing the argument \( x \) (see Cox [3]),

(ii) evaluating the non-zero B-splines of orders 1, 2, 3 and
4 by recurrence (see Cox [2] and [3]),

(iii) computing all derivatives of the B-splines of order 4
by applying a second recurrence to these computed B-spline
values (see de Boor [1]),

(iv) multiplying the 4th-order B-spline values and their
derivative by the appropriate B-spline coefficients, and
summing, to yield the values of \( s(x) \) and its derivatives.

E02BCF can be used to compute the values and derivatives of cubic
spline fits and interpolants produced by E02BAF.

If only values and not derivatives are required, E02BBF may be
used instead of E02BCF, which takes about 50% longer than E02BBF.

4. References


5. Parameters

1:  NCAP7  --  INTEGER
     Input
     
     On entry: n+7, where n is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range (\(\lambda\)) to (\(\lambda\_n+4\)) over which the spline is defined).

     Constraint: NCAP7 >= 8.

2:  LAMDA(NCAP7)  --  DOUBLE PRECISION array
     Input
     On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, (\(\lambda\_j\)), for \(j = 1, 2, \ldots, n+7\). Constraint: the LAMDA(j) must be in non-decreasing order with LAMDA(NCAP7-3) > LAMDA(4).

3:  C(NCAP7)  --  DOUBLE PRECISION array
     Input
     On entry: the coefficient c of the B-spline \(N_i(x)\), for \(i = 1, 2, \ldots, n+3\). The remaining elements of the array are not used.

4:  X  --  DOUBLE PRECISION
     Input
     On entry: the argument x at which the cubic spline and its derivatives are to be evaluated. Constraint: LAMDA(4) <= X <= LAMDA(NCAP7-3).

5:  LEFT  --  INTEGER
     Input
     On entry: specifies whether left- or right-hand values of the spline and its derivatives are to be computed (see Section 3). Left- or right-hand values are formed according
to whether LEFT is equal or not equal to 1. If x does not coincide with a knot, the value of LEFT is immaterial. If x = LAMDA(4), right-hand values are computed, and if x = LAMDA (NCAP7-3), left-hand values are formed, regardless of the value of LEFT.

6: S(4) -- DOUBLE PRECISION array Output
On exit: S(j) contains the value of the (j-1)th derivative of the spline at the argument x, for j = 1,2,3,4. Note that S(1) contains the value of the spline.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL= 2
Either LAMDA(4) >= LAMDA(NCAP7-3), i.e., the range over which s(x) is defined is null or negative in length, or X is an invalid argument, i.e., X < LAMDA(4) or X > LAMDA(NCAP7-3).

7. Accuracy

The computed value of s(x) has negligible error in most practical situations. Specifically, this value has an absolute error bounded in modulus by 18*c max max (j) the largest in modulus of c j ,c j+1 ,c j+2 , and c j+3 , and j is an integer such that (lambda) <=x<=(lambda) . If c j ,c j+1 ,c j+2 , and c j+3 are all of the same sign, then the computed value of j+3 s(x) has relative error bounded by 18*machine precision. For full details see Cox [3].

No complete error analysis is available for the computation of the derivatives of s(x). However, for most practical purposes the absolute errors in the computed derivatives should be small.
8. Further Comments

The time taken by this routine is approximately linear in

$$\log(n+7)$$.

Note: the routine does not test all the conditions on the knots
given in the description of LAMDA in Section 5, since to do this

would result in a computation time approximately linear in $$n+7$$

instead of $$\log(n+7)$$. All the conditions are tested in E02BAF, however.

9. Example

Compute, at the 7 arguments $$x = 0, 1, 2, 3, 4, 5, 6$$, the left-

and right-hand values and first 3 derivatives of the cubic spline
defined over the interval $$0 \leq x \leq 6$$ having the 6 interior knots $$x = 1, 3, 3, 3, 4, 4$$, the 8 additional knots $$0, 0, 0, 0, 6, 6, 6, 6$$,

and the 10 B-spline coefficients $$10, 12, 13, 15, 22, 26, 24, 18, 14, 12$$.

The input data items (using the notation of Section 5) comprise
the following values in the order indicated:

- $$n$$
- $$m$$
- $$\text{LAMDA}(j)$$, for $$j = 1, 2, \ldots, \text{NCAP7}$$
- $$\text{C}(j)$$, for $$j = 1, 2, \ldots, \text{NCAP7}-4$$
- $$x(i)$$, for $$i = 1, 2, \ldots, m$$

The example program is written in a general form that will enable
the values and derivatives of a cubic spline having an arbitrary
number of knots to be evaluated at a set of arbitrary points. Any
number of data sets may be supplied. The only changes required to
the program relate to the dimensions of the arrays LAMDA and C.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
E02 -- Curve and Surface Fitting
E02BDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BDF computes the definite integral of a cubic spline from its B-spline representation.

2. Specification

```
SUBROUTINE E02BDF (NCAP7, LAMDA, C, DEFINT, IFAIL)
  INTEGER NCAP7, IFAIL
  DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), DEFINT
```

3. Description

This routine computes the definite integral of the cubic spline \( s(x) \) between the limits \( x=a \) and \( x=b \), where \( a \) and \( b \) are respectively the lower and upper limits of the range over which \( s(x) \) is defined. It is assumed that \( s(x) \) is represented in terms of its B-spline coefficients \( c_i \), for \( i=1,2,\ldots,n+3 \) and

\[
\left( \text{augmented} \right) \text{ordered knot set} \ (\lambda_i) \ , \ \text{for} \ i=1,2,\ldots,n+7, \ \text{with} \ n+4, n+5, n+6, n+7, \ (\text{see E02BAF}), \ \text{i.e.,}
\]

\[
q -- \quad s(x) = \sum_{i=1}^{q} c_i N_i(x).
\]

Here \( q=n+3 \), \( n \) is the number of intervals of the spline and \( N_i(x) \)
denotes the normalised B-spline of degree 3 (order 4) defined upon the knots \( (\lambda_i) \), \( (\lambda_{i+1}) \), ..., \( (\lambda_{i+4}) \).

The method employed uses the formula given in Section 3 of Cox [1].

E02BDF can be used to determine the definite integrals of cubic spline fits and interpolants produced by E02BAF.

4. References


5. Parameters

1: NCAP7 -- INTEGER
   Input
   On entry: \( n+7 \), where \( n \) is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range \( a \) to \( b \)) over which the spline is defined. Constraint: \( NCAP7 \geq 8 \).

2: LAMDA(NCAP7) -- DOUBLE PRECISION array
   Input
   On entry: \( LAMDA(j) \) must be set to the value of the \( j \)th member of the complete set of knots, \( (\lambda_j) \) for \( j=1,2,...,n+7 \). Constraint: \( LAMDA(j) \) must be in non-decreasing order with \( LAMDA(NCAP7-3) > LAMDA(4) \) and satisfy
   \( LAMDA(1)=LAMDA(2)=LAMDA(3)=LAMDA(4) \)
   and
   \( LAMDA(NCAP7-3)=LAMDA(NCAP7-2)=LAMDA(NCAP7-1)=LAMDA(NCAP7) \).

3: C(NCAP7) -- DOUBLE PRECISION array
   Input
   On entry: the coefficient \( c \) of the B-spline \( N_i(x) \), for \( i=1,2,...,n+3 \). The remaining elements of the array are not used.

4: DEFINT -- DOUBLE PRECISION
   Output
   On exit: the value of the definite integral of \( s(x) \) between the limits \( x=a \) and \( x=b \), where \( a=(\lambda_4) \) and \( b=(\lambda_{NCAP7}) \).
5: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL = 2
   At least one of the following restrictions on the knots is
   violated:
   LAMDA(NCAP7-3) > LAMDA(4),
   LAMDA(j) >= LAMDA(j-1),
   for j = 2,3,...,NCAP7, with equality in the cases
   j=2,3,4,NCAP7-2,NCAP7-1, and NCAP7.

7. Accuracy

The rounding errors are such that the computed value of the
integral is exact for a slightly perturbed set of B-spline
coefficients c differing in a relative sense from those supplied
by no more than 2.2*(n+3)*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to

n+7.

9. Example

Determine the definite integral over the interval 0<=x<=6 of a
cubic spline having 6 interior knots at the positions (lambda)=1,
3, 3, 3, 4, 4, the 8 additional knots 0, 0, 0, 0, 6, 6, 6, 6, and
the 10 B-spline coefficients 10, 12, 13, 15, 22, 26, 24, 18, 14,
12.

The input data items (using the notation of Section 5) comprise the following values in the order indicated:

\[ n \]
\[ \text{LAMDA}(j) \text{ for } j = 1,2,...,\text{NCAP7} \]
\[ \text{C}(j), \text{ for } j = 1,2,...,\text{NCAP7}-3 \]

The example program is written in a general form that will enable the definite integral of a cubic spline having an arbitrary number of knots to be computed. Any number of data sets may be supplied. The only changes required to the program relate to the dimensions of the arrays LAMDA and C.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting
E02BEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BEF computes a cubic spline approximation to an arbitrary set of data points. The knots of the spline are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

2. Specification

SUBROUTINE E02BEF (START, M, X, Y, W, S, NEST, N, LAMDA, C, FP, WRK, LWRK, IWRK, IFAIL)
INTEGER M, NEST, N, LWRK, IWRK(NEST), IFAIL
DOUBLE PRECISION X(M), Y(M), W(M), S, LAMDA(NEST), C(NEST), FP, WRK(LWRK)
CHARACTER*1 START
3. Description

This routine determines a smooth cubic spline approximation \( s(x) \) to the set of data points \((x_r, y_r)\), with weights \( w_r\), for \( r = 1, 2, \ldots, m\).

The spline is given in the B-spline representation

\[
s(x) = \sum_{i=1}^{n-4} c_i N_i^{(3)}(x)
\]

where \( N_i^{(3)}(x) \) denotes the normalised cubic B-spline defined upon the knots \( \lambda_i, \lambda_{i+1}, \ldots, \lambda_{i+4} \).

The total number \( n \) of these knots and their values \( \lambda_1, \ldots, \lambda_n \) are chosen automatically by the routine.

The knots \( \lambda_1, \ldots, \lambda_n \) are the interior knots; they divide the approximation interval \([x_1, x_m]\) into \( n-7 \) sub-intervals.

The coefficients \( c_1, c_2, \ldots, c_{n-4} \) are then determined as the solution of the following constrained minimization problem:

minimize

\[
\sum_{i=5}^{n-4} \delta_i^2
\]

subject to the constraint

\[
\sum_{r=1}^{m} \epsilon_r \leq S
\]

where: \( \delta_i \) stands for the discontinuity jump in the third \( i \) order derivative of \( s(x) \) at the interior knot.
i

(\epsilon) denotes the weighted residual \( w(y - s(x)) \),

\[ r \]

and \( S \) is a non-negative number to be specified by
the user.

The quantity (\eta) can be seen as a measure of the (lack of)
smoothness of \( s(x) \), while closeness of fit is measured through
(\theta). By means of the parameter \( S \), 'the smoothing factor', the
user will then control the balance between these two (usually
conflicting) properties. If \( S \) is too large, the spline will be
too smooth and signal will be lost (underfit); if \( S \) is too small,
the spline will pick up too much noise (overfit). In the extreme
cases the routine will return an interpolating spline ((\theta)=0)
if \( S \) is set to zero, and the weighted least-squares cubic
polynomial ((\eta)=0) if \( S \) is set very large. Experimenting with \( S \)
values between these two extremes should result in a good
compromise. (See Section 8.2 for advice on choice of \( S \).)

The method employed is outlined in Section 8.3 and fully
described in Dierckx [1], [2] and [3]. It involves an adaptive
strategy for locating the knots of the cubic spline (depending on
the function underlying the data and on the value of \( S \)), and an
iterative method for solving the constrained minimization problem
once the knots have been determined.

Values of the computed spline, or of its derivatives or definite
integral, can subsequently be computed by calling E02BBF, E02BCF
or E02BDF, as described in Section 8.4.

4. References

and Integration of Experimental Data Using Spline Functions.

Rectangular Grid while using Spline Functions. SIAM J.

with Spline Functions. Report TW54. Department of Computer
Science, Katholieke Universiteit Leuven.

10 177--183.
5. Parameters

1: START -- CHARACTER*1  
   Input
   On entry: START must be set to 'C' or 'W'.

   If START = 'C' (Cold start), the routine will build up the
   knot set starting with no interior knots. No values need be
   assigned to the parameters N, LAMDA, WRK or IWRK.

   If START = 'W' (Warm start), the routine will restart the
   knot-placing strategy using the knots found in a previous
   call of the routine. In this case, the parameters N, LAMDA,
   WRK, and IWRK must be unchanged from that previous call.
   This warm start can save much time in searching for a
   satisfactory value of S. Constraint: START = 'C' or 'W'.

2: M -- INTEGER  
   Input
   On entry: m, the number of data points. Constraint: M >= 4.

3: X(M) -- DOUBLE PRECISION array  
   Input
   On entry: the values x of the independent variable
   \( x_r \) for \( r=1,2,\ldots,m \). Constraint: \( x_1 < x_2 < \ldots < x_m \)

4: Y(M) -- DOUBLE PRECISION array  
   Input
   On entry: the values y of the dependent variable
   \( y_r \) for \( r=1,2,\ldots,m \).

5: W(M) -- DOUBLE PRECISION array  
   Input
   On entry: the values w of the weights, for \( r=1,2,\ldots,m \).
   For advice on the choice of weights, see the Chapter
   Introduction, Section 2.1.2. Constraint: \( W(r) > 0 \), for
   \( r=1,2,\ldots,m \).

6: S -- DOUBLE PRECISION  
   Input
   On entry: the smoothing factor, S.

   If S=0.0, the routine returns an interpolating spline.

   If S is smaller than machine precision, it is assumed equal
to zero.

   For advice on the choice of S, see Section 3 and Section 8.2
   Constraint: \( S \geq 0.0 \).

7: NEST -- INTEGER  
   Input
   On entry: an over-estimate for the number, n, of knots
required. Constraint: \( N \geq 8 \). In most practical situations, \( N = M/2 \) is sufficient. \( N \) never needs to be larger than \( M + 4 \), the number of knots needed for interpolation \( (S = 0.0) \).

8: \( N \) -- INTEGER
   Input/Output
   On entry: if the warm start option is used, the value of \( N \)
   must be left unchanged from the previous call. On exit: the
   total number, \( n \), of knots of the computed spline.

9: \( \text{LAMDA(NEST)} \) -- DOUBLE PRECISION array
   Input/Output
   On entry: if the warm start option is used, the values
   \( \text{LAMDA}(1), \text{LAMDA}(2),\ldots,\text{LAMDA}(N) \) must be left unchanged from
   the previous call. On exit: the knots of the spline i.e.,
   the positions of the interior knots \( \text{LAMDA}(5), \text{LAMDA}(6),\ldots \)
   ,\( \text{LAMDA}(N-4) \) as well as the positions of the additional knots
   \( \text{LAMDA}(1) = \text{LAMDA}(2) = \text{LAMDA}(3) = \text{LAMDA}(4) = x \) and
   \( \text{LAMDA}(N-3) = \text{LAMDA}(N-2) = \text{LAMDA}(N-1) = \text{LAMDA}(N) = x \) needed
   for the B-spline representation.

10: \( C(NEST) \) -- DOUBLE PRECISION array
    Output
    On exit: the coefficient \( c \) of the B-spline \( N(x) \) in the
    spline approximation \( s(x) \), for \( i=1,2,\ldots,n-4 \).

11: \( FP \) -- DOUBLE PRECISION
    Output
    On exit: the sum of the squared weighted residuals, \( (theta) \),
    of the computed spline approximation. If \( FP = 0.0 \), this is
    an interpolating spline. \( FP \) should equal \( S \) within a relative
    tolerance of 0.001 unless \( n=8 \) when the spline has no
    interior knots and so is simply a cubic polynomial. For
    knots to be inserted, \( S \) must be set to a value below the
    value of \( FP \) produced in this case.

12: \( \text{WRK(LWRK)} \) -- DOUBLE PRECISION array
    Workspace
    On entry: if the warm start option is used, the values \( \text{WRK}(1),\ldots,\text{WRK}(n) \) must be left unchanged from the previous
    call.

13: \( LWRK \) -- INTEGER
    Input
    On entry:
    the dimension of the array \( \text{WRK} \) as declared in the
    (sub)program from which \( \text{E02BEF} \) is called.
    Constraint: \( LWRK \geq 4*M+16*NEST+41 \).

14: \( \text{IWRK(NEST)} \) -- INTEGER array
    Workspace
    On entry: if the warm start option is used, the values \( \text{IWRK} \)
(1), ..., IWRK(n) must be left unchanged from the previous call.

This array is used as workspace.

15: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
    On entry START /= 'C' or 'W',
    or M < 4,
    or S < 0.0,
    or S = 0.0 and NEST < M + 4,
    or NEST < 8,
    or LWRK<4*M+16*NEST+41.

IFAIL= 2
    The weights are not all strictly positive.

IFAIL= 3
    The values of X(r), for r=1,2,...,M, are not in strictly increasing order.

IFAIL= 4
    The number of knots required is greater than NEST. Try increasing NEST and, if necessary, supplying larger arrays for the parameters LAMDA, C, WRK and IWRK. However, if NEST is already large, say NEST > M/2, then this error exit may indicate that S is too small.

IFAIL= 5
    The iterative process used to compute the coefficients of the approximating spline has failed to converge. This error
exit may occur if $S$ has been set very small. If the error persists with increased $S$, consult NAG.

If IFAIL = 4 or 5, a spline approximation is returned, but it fails to satisfy the fitting criterion (see (2) and (3) in Section 3) – perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its weighted sum of squared residuals $FP$ is equal to the smoothing factor $S$, up to a specified relative tolerance of 0.001 – except that if $n=8$, $FP$ may be significantly less than $S$: in this case the computed spline is simply a weighted least-squares polynomial approximation of degree 3, i.e., a spline with no interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02BEF depends on the complexity of the shape of the data, the value of the smoothing factor $S$, and the number of data points. If E02BEF is to be called for different values of $S$, much time can be saved by setting START =

8.2. Choice of $S$

If the weights have been correctly chosen (see Section 2.1.2 of the Chapter Introduction), the standard deviation of $w_y$ would be the same for all $r$, equal to $(\sigma)$, say. In this case, choosing the smoothing factor $S$ in the range $(\sigma) (m-\sqrt{2}m)$, as suggested by Reinsch [4], is likely to give a good start in the search for a satisfactory value. Otherwise, experimenting with different values of $S$ will be required from the start, taking account of the remarks in Section 3.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for $S$ and so determine the least-squares cubic polynomial; the value returned for $FP$, call it $FP_0$, gives an upper bound for $S_0$.

Then progressively decrease the value of $S$ to obtain closer fits – say by a factor of 10 in the beginning, i.e., $S=FP_0/10$, $S=FP_0/100$, and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location,
generally depend on the value of \( S \) and on the behaviour of the function underlying the data. However, if \( E02BEF \) is called with \( START = 'W' \), the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of \( S \) and \( START = 'W' \), a fit can finally be accepted as satisfactory, it may be worthwhile to call \( E02BEF \) once more with the selected value for \( S \) but now using \( START = 'C' \). Often, \( E02BEF \) then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Outline of Method Used

If \( S = 0 \), the requisite number of knots is known in advance, i.e., \( n = m + 4 \); the interior knots are located immediately as \( (\lambda_i) = x_i \) for \( i = 5, 6, \ldots, n - 4 \). The corresponding least-squares spline (see \( E02BAF \)) is then an interpolating spline and therefore a solution of the problem.

If \( S > 0 \), a suitable knot set is built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a spline is fitted to the data by least-squares (see \( E02BAF \)) and \( (\theta) \), the weighted sum of squares of residuals, is computed. If \( (\theta) > S \), new knots are added to the knot set to reduce \( (\theta) \) at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of \( S \) and on the progress made so far in reducing \( (\theta) \). Sooner or later, we find that \( (\theta) <= S \) and at that point the knot set is accepted. The routine then goes on to compute the (unique) spline which has this knot set and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has \( (\theta) = S \). The routine computes the spline by an iterative scheme which is ended when \( (\theta) = S \) within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in \( E02BAF \).

An exception occurs when the routine finds at the start that, even with no interior knots \( (n = 8) \), the least-squares spline already has its weighted sum of squares of residuals \( <= S \). In this case, since this spline (which is simply a cubic polynomial) also has an optimal value for the smoothness measure \( (\eta) \), namely zero, it is returned at once as the (trivial) solution. It will usually mean that \( S \) has been chosen too large.

For further details of the algorithm and its use, see Dierckx [3]
8.4. Evaluation of Computed Spline

The value of the computed spline at a given value $X$ may be obtained in the double precision variable $S$ by the call:

```
CALL E02BBF(N,LAMDA,C,X,S,IFAIL)
```

where $N$, LAMDA and C are the output parameters of E02BEF.

The values of the spline and its first three derivatives at a given value $X$ may be obtained in the double precision array SDIF of dimension at least 4 by the call:

```
CALL E02BCF(N,LAMDA,C,X,LEFT,SDIF,IFAIL)
```

where if LEFT = 1, left-hand derivatives are computed and if LEFT /= 1, right-hand derivatives are calculated. The value of LEFT is only relevant if $X$ is an interior knot.

The value of the definite integral of the spline over the interval $X(1)$ to $X(M)$ can be obtained in the double precision variable SINT by the call:

```
CALL E02BDF(N,LAMDA,C,SINT,IFAIL)
```

9. Example

This example program reads in a set of data values, followed by a set of values of $S$. For each value of $S$ it calls E02BEF to compute a spline approximation, and prints the values of the knots and the $B$-spline coefficients $c_i$.

The program includes code to evaluate the computed splines, by calls to E02BBF, at the points $x$ and at points mid-way between them. These values are not printed out, however; instead the results are illustrated by plots of the computed splines, together with the data points (indicated by *) and the positions of the knots (indicated by vertical lines): the effect of decreasing $S$ can be clearly seen. (The plots were obtained by calling NAG Graphical Supplement routine J06FAF(*).)

```
Please see figures in printed Reference Manual
```

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation
1. Purpose

E02DAF forms a minimal, weighted least-squares bicubic spline surface fit with prescribed knots to a given set of data points.

2. Specification

```
SUBROUTINE E02DAF (M, PX, PY, X, Y, F, W, LAMDA, MU,
  1       POINT, NPOINT, DL, C, NC, WS, NWS, EPS,
  2       SIGMA, RANK, IFAIL)
INTEGER M, PX, PY, POINT(NPOINT), NPOINT, NC, NWS,
  1       RANK, IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), W(M), LAMDA(PX), MU(PY),
  1       DL(NC), C(NC), WS(NWS), EPS, SIGMA
```

3. Description

This routine determines a bicubic spline fit \( s(x,y) \) to the set of data points \((x_r, y_r, f_r)\) with weights \( w_r\), for \( r=1,2,...,m \). The two sets of internal knots of the spline, \( \{(\lambda)\} \) and \( \{(\mu)\} \), associated with the variables \( x \) and \( y \) respectively, are prescribed by the user. These knots can be thought of as dividing the data region of the \((x,y)\) plane into panels (see diagram in Section 5). A bicubic spline consists of a separate bicubic polynomial in each panel, the polynomials joining together with continuity up to the second derivative across the panel boundaries.

\( s(x,y) \) has the property that \( (\text{Sigma}) \), the sum of squares of its weighted residuals \( (\rho) \), for \( r=1,2,...,m \), where

\[
(\rho_r) = w_r (s(x_r, y_r) - f_r),
\]

is as small as possible for a bicubic spline with the given knot sets. The routine produces this minimized value of \( (\text{Sigma}) \) and
the coefficients \( c_{ij} \) in the B-spline representation of \( s(x,y) \) - see Section 8. E02DEF and E02DFF are available to compute values of the fitted spline from the coefficients \( c_{ij} \).

The least-squares criterion is not always sufficient to determine the bicubic spline uniquely: there may be a whole family of splines which have the same minimum sum of squares. In these cases, the routine selects from this family the spline for which the sum of squares of the coefficients \( c_{ij} \) is smallest: in other words, the minimal least-squares solution. This choice, although arbitrary, reduces the risk of unwanted fluctuations in the spline fit. The method employed involves forming a system of \( m \) linear equations in the coefficients \( c_{ij} \) and then computing its least-squares solution, which will be the minimal least-squares solution when appropriate. The basis of the method is described in Hayes and Halliday [4]. The matrix of the equation is formed using a recurrence relation for B-splines which is numerically stable (see Cox [1] and de Boor [2] - the former contains the more elementary derivation but, unlike [2], does not cover the case of coincident knots). The least-squares solution is also obtained in a stable manner by using orthogonal transformations, viz. a variant of Givens rotation (see Gentleman [3]). This requires only one row of the matrix to be stored at a time. Advantage is taken of the stepped-band structure which the matrix possesses when the data points are suitably ordered, there being at most sixteen non-zero elements in any row because of the definition of B-splines. First the matrix is reduced to upper triangular form and then the diagonal elements of this triangle are examined in turn. When an element is encountered whose square, divided by the mean squared weight, is less than a threshold (epsilon), it is replaced by zero and the rest of the elements in its row are reduced to zero by rotations with the remaining rows. The rank of the system is taken to be the number of non-zero diagonal elements in the final triangle, and the non-zero rows of this triangle are used to compute the minimal least-squares solution. If all the diagonal elements are non-zero, the rank is equal to the number of coefficients \( c_{ij} \) and the solution obtained is the ordinary least-squares solution, which is unique in this case.

4. References

5. Parameters

1: M -- INTEGER
   On entry: the number of data points, m. Constraint: M > 1.

2: PX -- INTEGER

3: PY -- INTEGER
   On entry: the total number of knots (\(\lambda\)) and (\(\mu\))
   associated with the variables \(x\) and \(y\), respectively.
   Constraint: PX >= 8 and PY >= 8.
   (They are such that PX-8 and PY-8 are the corresponding
   numbers of interior knots.) The running time and storage
   required by the routine are both minimized if the axes are
   labelled so that PY is the smaller of PX and PY.

4: X(M) -- DOUBLE PRECISION array

5: Y(M) -- DOUBLE PRECISION array

6: F(M) -- DOUBLE PRECISION array
   On entry: the co-ordinates of the data point \((x_r, y_r, f_r)\), for
   \(r = 1, 2, \ldots, m\). The order of the data points is immaterial, but
   see the array POINT, below.

7: W(M) -- DOUBLE PRECISION array
   On entry: the weight \(w_r\) of the \(r\)th data point. It is
   important to note the definition of weight implied by the
   equation (1) in Section 3, since it is also common usage to
   define weight as the square of this weight. In this routine,
   each \(w\) should be chosen inversely proportional to the
   (absolute) accuracy of the corresponding \(f\), as expressed,
   for example, by the standard deviation or probable error of
   the \(f\). When the \(f\) are all of the same accuracy, all the \(w\)
may be set equal to 1.0.

8: LAMDA(PX) -- DOUBLE PRECISION array Input/Output
On entry: LAMDA(i+4) must contain the ith interior knot (\(\lambda\)) associated with the variable \(x\), for
\[i+4\]
i=1,2,...,PX-8. The knots must be in non-decreasing order and lie strictly within the range covered by the data values of \(x\). A knot is a value of \(x\) at which the spline is allowed to be discontinuous in the third derivative with respect to \(x\), though continuous up to the second derivative. This degree of continuity can be reduced, if the user requires, by the use of coincident knots, provided that no more than four knots are chosen to coincide at any point. Two, or three, coincident knots allow loss of continuity in, respectively, the second and first derivative with respect to \(x\) at the value of \(x\) at which they coincide. Four coincident knots split the spline surface into two independent parts. For choice of knots see Section 8. On exit: the interior knots LAMDA(5) to LAMDA(PX-4) are unchanged, and the segments LAMDA(1:4) and LAMDA(PX-3:PX) contain additional (exterior) knots introduced by the routine in order to define the full set of B-splines required. The four knots in the first segment are all set equal to the lowest data value of \(x\) and the other four additional knots are all set equal to the highest value: there is experimental evidence that coincident end-knots are best for numerical accuracy. The complete array must be left undisturbed if E02DEF or E02DFF is to be used subsequently.

9: MU(PY) -- DOUBLE PRECISION array Input
On entry: MU(i+4) must contain the ith interior knot (\(\mu\)) associated with the variable \(y\), i=1,2,...,PY-8. The same remarks apply to MU as to LAMDA above, with \(Y\) replacing \(X\), and \(y\) replacing \(x\).

10: POINT(NPOINT) -- INTEGER array Input
On entry: indexing information usually provided by E02ZAF which enables the data points to be accessed in the order which produces the advantageous matrix structure mentioned in Section 3. This order is such that, if the \((x,y)\) plane is thought of as being divided into rectangular panels by the two sets of knots, all data in a panel occur before data in succeeding panels, where the panels are numbered from bottom to top and then left to right with the usual arrangement of axes, as indicated in the diagram.

Please see figure in printed Reference Manual
A data point lying exactly on one or more panel sides is considered to be in the highest numbered panel adjacent to the point. E02ZAF should be called to obtain the array POINT, unless it is provided by other means.

11: NPOINT -- INTEGER
   Input
   On entry:
   the dimension of the array POINT as declared in the (sub)program from which E02DAF is called.
   Constraint: NPOINT >= M + (PX-7)*(PY-7).

12: DL(NC) -- DOUBLE PRECISION array
    Output
    On exit: DL gives the squares of the diagonal elements of the reduced triangular matrix, divided by the mean squared weight. It includes those elements, less than (epsilon), which are treated as zero (see Section 3).

13: C(NC) -- DOUBLE PRECISION array
    Output
    On exit: C gives the coefficients of the fit. C((PY-4)*(i-1)+j) is the coefficient of Section 3 and Section 8 for
    \[ i=1,2,...,PX-4 \text{ and } j=1,2,...,PY-4. \] These coefficients are used by E02DEF or E02DFF to calculate values of the fitted function.

14: NC -- INTEGER
    Input
    On entry: the value (PX-4)*(PY-4).

15: WS(NWS) -- DOUBLE PRECISION array
    Workspace

16: NWS -- INTEGER
    Input
    On entry:
    the dimension of the array WS as declared in the (sub)program from which E02DAF is called.
    Constraint: NWS>=((2*NC+1)*(3*PY-6)-2).

17: EPS -- DOUBLE PRECISION
    Input
    On entry: a threshold (epsilon) for determining the effective rank of the system of linear equations. The rank is determined as the number of elements of the array DL (see below) which are non-zero. An element of DL is regarded as zero if it is less than (epsilon). Machine precision is a suitable value for (epsilon) in most practical applications which have only 2 or 3 decimals accurate in data. If some coefficients of the fit prove to be very large compared with the data ordinates, this suggests that (epsilon) should be increased so as to decrease the rank. The array DL will give a guide to appropriate values of (epsilon) to achieve this, as well as to the choice of (epsilon) in other cases where
some experimentation may be needed to determine a value which leads to a satisfactory fit.

18: SIGMA -- DOUBLE PRECISION  Output
On exit: (Sigma), the weighted sum of squares of residuals. This is not computed from the individual residuals but from the right-hand sides of the orthogonally-transformed linear equations. For further details see Hayes and Halliday [4] page 97. The two methods of computation are theoretically equivalent, but the results may differ because of rounding error.

19: RANK -- INTEGER  Output
On exit: the rank of the system as determined by the value of the threshold (epsilon). When RANK = NC, the least-squares solution is unique: in other cases the minimal least-squares solution is computed.

20: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one set of knots is not in non-decreasing order, or an interior knot is outside the range of the data values.

IFAIL= 2
More than four knots coincide at a single point, possibly because all data points have the same value of x (or y) or because an interior knot coincides with an extreme data value.

IFAIL= 3
Array POINT does not indicate the data points in panel order. Call E02ZAF to obtain a correct array.

IFAIL= 4
On entry M <= 1,
  or       PX < 8,
  or       PY < 8,
or \( NC /= (PX-4)*(PY-4), \)
or NWS is too small,
or NPOINT is too small.

IFAIL= 5
All the weights \( w \) are zero or rank determined as zero.

7. Accuracy

The computation of the B-splines and reduction of the observation matrix to triangular form are both numerically stable.

8. Further Comments

The time taken by this routine is approximately proportional to 

\[ \frac{7}{2} \]

the number of data points, \( m \), and to \( (3*(PY-4)+4) \).

The B-spline representation of the bicubic spline is

\[
-\sum_{ij} c_{ij} M_i(x) N_j(y)
\]

summed over \( i=1,2,...,PX-4 \) and over \( j=1,2,...,PY-4 \). Here \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the former defined on the knots \( \lambda_i, \lambda_{i+1}, \lambda_{i+4} \) and the latter on the knots \( \mu_j, \mu_{j+1}, \mu_{j+4} \). For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [2] for normalised B-splines.

The choice of the interior knots, which help to determine the spline’s shape, must largely be a matter of trial and error. It is usually best to start with a small number of knots and, examining the fit at each stage, add a few knots at a time at places where the fit is particularly poor. In intervals of \( x \) or \( y \) where the surface represented by the data changes rapidly, in function value or derivatives, more knots will be needed than elsewhere. In some cases guidance can be obtained by analogy with the case of coincident knots: for example, just as three coincident knots can produce a discontinuity in slope, three
close knots can produce rapid change in slope. Of course, such rapid changes in behaviour must be adequately represented by the data points, as indeed must the behaviour of the surface generally, if a satisfactory fit is to be achieved. When there is no rapid change in behaviour, equally-spaced knots will often suffice.

In all cases the fit should be examined graphically before it is accepted as satisfactory.

The fit obtained is not defined outside the rectangle

\[ \lambda \leq x \leq \lambda, \quad \mu \leq y \leq \mu \]

The reason for taking the extreme data values of \( x \) and \( y \) for these four knots is that, as is usual in data fitting, the fit cannot be expected to give satisfactory values outside the data region. If, nevertheless, the user requires values over a larger rectangle, this can be achieved by augmenting the data with two artificial data points \((a,c,0)\) and \((b,d,0)\) with zero weight, where \( a \leq x \leq b, \quad c \leq y \leq d \) defines the enlarged rectangle. In the case when the data are adequate to make the least-squares solution unique (\( RANK = NC \)), this enlargement will not affect the fit over the original rectangle, except for possibly enlarged rounding errors, and will simply continue the bicubic polynomials in the panels bordering the rectangle out to the new boundaries: in other cases the fit will be affected. Even using the original rectangle there may be regions within it, particularly at its corners, which lie outside the data region and where, therefore, the fit will be unreliable. For example, if there is no data point in panel 1 of the diagram in Section 5, the least-squares criterion leaves the spline indeterminate in this panel: the minimal spline determined by the subroutine in this case passes through the value zero at the point \((\lambda, \mu)\).

9. Example

This example program reads a value for \((\epsilon)\), and a set of data points, weights and knot positions. If there are more \( y \) knots than \( x \) knots, it interchanges the \( x \) and \( y \) axes. It calls \texttt{E02ZAF} to sort the data points into panel order, \texttt{E02DAF} to fit a bicubic spline to them, and \texttt{E02DEF} to evaluate the spline at the data points.

Finally it prints:

the weighted sum of squares of residuals computed from the linear equations;
The rank determined by E02DAF;

data points, fitted values and residuals in panel order;

the weighted sum of squares of the residuals;

the coefficients of the spline fit.

The program is written to handle any number of data sets.

Note: the data supplied in this example is not typical of a
realistic problem: the number of data points would normally be
much larger (in which case the array dimensions and the value of
NWS in the program would have to be increased); and the value of
(epsilon) would normally be much smaller on most machines (see
Section 5; the relatively large value of 10^-6 has been chosen in
order to illustrate a minimal least-squares solution when RANK <
NC; in this example NC = 24).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting

E02DCF -- NAG Foundation Library Routine Document

SUBROUTINE E02DCF (START, MX, X, MY, Y, F, S, NXEST, 1
1 NYEST, NX, LAMDA, NY, MU, C, FP, WRK, 2
2 LWRK, LWRK, LIWRK, IFAIL)

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DCF computes a bicubic spline approximation to a set of data
values, given on a rectangular grid in the x-y plane. The knots
of the spline are located automatically, but a single parameter
must be specified to control the trade-off between closeness of
fit and smoothness of fit.

2. Specification
3. Description

This routine determines a smooth bicubic spline approximation $s(x,y)$ to the set of data points $(x_q,y_r,f_{qr})$, for $q=1,2,...,m$ and $r=1,2,...,m$.

The spline is given in the B-spline representation

$$s(x,y) = \sum_{i=1}^{n-4} \sum_{j=1}^{n-4} c_{ij} M_i(x) N_j(y),$$

where $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines, the former defined on the knots $(\lambda_i)$ to $(\lambda_{i+4})$ and the latter on the knots $(\mu_j)$ to $(\mu_{j+4})$. For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [1] for normalised B-splines.

The total numbers $n$ and $n$ of these knots and their values $(\lambda_1,\ldots,\lambda_n)$ and $(\mu_1,\ldots,\mu_n)$ are chosen automatically by the routine. The knots $(\lambda_1),\ldots,5$ $(\lambda_{n-4})$ and $(\mu_1),\ldots,(\mu_{n-4})$ are the interior knots; they divide the approximation domain $[x_1,x_m] \times [y_1,y_m]$ into $m \times m$ subpanels $[(\lambda_i),(\lambda_{i+1})] \times [(\mu_j),(\mu_{j+1})]$ for $i=4,5,\ldots,n-4$, $j=4,5,\ldots,n-4$. Then, much as in the curve case (see E02BEF), the coefficients $c_{ij}$ are determined as the...
solution of the following constrained minimization problem:

minimize

\[ (\eta), \quad (2) \]

subject to the constraint

\[ \begin{array}{ccc}
  m & m & 2 \\
  x & y & = \\
  -- & -- & \theta \\
  q & r & \leq S, \\
  q=1 & r=1 \\
\end{array} \quad (3) \]

where \((\eta)\) is a measure of the (lack of) smoothness of \(s(x,y)\).
Its value depends on the discontinuity jumps in \(s(x,y)\) across the boundaries of the subpanels. It is zero only when there are no discontinuities and is positive otherwise, increasing with the size of the jumps (see Dierckx [2] for details).

\((\epsilon)\) denotes the residual \(f - s(x,y)\),

\[ q,r \quad q,r \]

and \(S\) is a non-negative number to be specified by the user.

By means of the parameter \(S\), 'the smoothing factor', the user will then control the balance between smoothness and closeness of fit, as measured by the sum of squares of residuals in (3). If \(S\) is too large, the spline will be too smooth and signal will be lost (underfit); if \(S\) is too small, the spline will pick up too much noise (overfit). In the extreme cases the routine will return an interpolating spline ((\(\eta) = 0\)) if \(S\) is set to zero, and the least-squares bicubic polynomial ((\(\eta) = 0\)) if \(S\) is set very large. Experimenting with \(S\)-values between these two extremes should result in a good compromise. (See Section 8.3 for advice on choice of \(S\).)

The method employed is outlined in Section 8.5 and fully described in Dierckx [2] and [3]. It involves an adaptive strategy for locating the knots of the bicubic spline (depending on the function underlying the data and on the value of \(S\)), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline can subsequently be computed by calling E02DEF or E02DFF as described in Section 8.6.

4. References
5. Parameters

1: \texttt{START} -- \texttt{CHARACTER*1} \hspace{1cm} \texttt{Input}
\hspace{1cm} On entry: \texttt{START} must be set to 'C' or 'W'.
\hspace{1cm} If \texttt{START} = 'C' (Cold start), the routine will build up the
\hspace{1cm} knot set starting with no interior knots. No values need be
\hspace{1cm} assigned to the parameters \texttt{NX}, \texttt{NY}, \texttt{LAMDA}, \texttt{MU}, \texttt{WRK} or \texttt{IWRK}.
\hspace{1cm} If \texttt{START} = 'W' (Warm start), the routine will restart the
\hspace{1cm} knot-placing strategy using the knots found in a previous
\hspace{1cm} call of the routine. In this case, the parameters \texttt{NX}, \texttt{NY},
\hspace{1cm} \texttt{LAMDA}, \texttt{MU}, \texttt{WRK} and \texttt{IWRK} must be unchanged from that previous
\hspace{1cm} call. This warm start can save much time in searching for a
\hspace{1cm} satisfactory value of \texttt{S}. Constraint: \texttt{START} = 'C' or 'W'.

2: \texttt{MX} -- \texttt{INTEGER} \hspace{1cm} \texttt{Input}
\hspace{1cm} On entry: \texttt{m}, the number of grid points along the \texttt{x} axis.
\hspace{1cm} \texttt{x}
\hspace{1cm} Constraint: \texttt{MX} \geq 4.

3: \texttt{X(MX)} -- \texttt{DOUBLE PRECISION array} \hspace{1cm} \texttt{Input}
\hspace{1cm} On entry: \texttt{X(q)} must be set to \texttt{x}, the \texttt{x} co-ordinate of the
\hspace{1cm} \texttt{qth} grid point along the \texttt{x} axis, for \texttt{q=1,2,...,m}.
\hspace{1cm} \texttt{x}
\hspace{1cm} Constraint: \texttt{x} < \texttt{x} < \ldots < \texttt{x}.
\hspace{1cm} 1 2 \cdots \texttt{m}
\hspace{1cm} \texttt{x}

4: \texttt{MY} -- \texttt{INTEGER} \hspace{1cm} \texttt{Input}
On entry: \( m \), the number of grid points along the \( y \) axis. 
\( \text{Constraint: } MY \geq 4. \)

5: \( Y(MY) \) -- DOUBLE PRECISION array 
\( \text{Input} \)
On entry: \( Y(r) \) must be set to \( y \), the \( y \) co-ordinate of the \( r \)th grid point along the \( y \) axis, for \( r=1,2,\ldots,m \).
\( \text{Constraint: } y < y < \ldots < y \).
\( \quad 1 \quad 2 \quad m \)
\( y \)

6: \( F(MX*MY) \) -- DOUBLE PRECISION array 
\( \text{Input} \)
On entry: \( F(m *(q-1)+r) \) must contain the data value \( f \), for \( q=1,2,\ldots,m \) and \( r=1,2,\ldots,m \).
\( \text{Constraint: } S \geq 0.0. \)

7: \( S \) -- DOUBLE PRECISION 
\( \text{Input} \)
On entry: the smoothing factor, \( S \).
If \( S=0.0 \), the routine returns an interpolating spline.
If \( S \) is smaller than machine precision, it is assumed equal to zero.
For advice on the choice of \( S \), see Section 3 and Section 8.3.
\( \text{Constraint: } S \geq 0.0. \)

8: \( NXEST \) -- INTEGER 
\( \text{Input} \)

9: \( NYEST \) -- INTEGER 
\( \text{Input} \)
On entry: an upper bound for the number of knots \( n \) and \( n \) required in the \( x \)- and \( y \)-directions respectively.

In most practical situations, \( NXEST = m /2 \) and \( NYEST = m /2 \) is sufficient. \( NXEST \) and \( NYEST \) never need to be larger than \( m +4 \) and \( m +4 \) respectively, the numbers of knots needed for interpolation (\( S=0.0 \)). See also Section 8.4. \( \text{Constraint: } NXEST \geq 8 \) and \( NYEST \geq 8. \)

10: \( NX \) -- INTEGER 
\( \text{Input/Output} \)
On entry: if the warm start option is used, the value of \( NX \) must be left unchanged from the previous call. On exit: the total number of knots, \( n \), of the computed spline with \( x \).
respect to the x variable.

11: LAMDA(NXEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values
LAMDA(1), LAMDA(2),...,LAMDA(NX) must be left unchanged from
the previous call. On exit: LAMDA contains the complete set
of knots (lambda) associated with the x variable, i.e., the
interior knots LAMDA(5), LAMDA(6), ..., LAMDA(NX-4) as well
as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) =
LAMDA(4) = X(1) and LAMDA(NX-3) = LAMDA(NX-2) = LAMDA(NX-1) =
LAMDA(NX) = X(MX) needed for the B-spline representation.

12: NY -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NY
must be left unchanged from the previous call. On exit: the
total number of knots, n, of the computed spline with
respect to the y variable.

13: MU(NYEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values MU
(1), MU(2),...,MU(NY) must be left unchanged from the
previous call. On exit: MU contains the complete set of
knots (mu) associated with the y variable, i.e., the
interior knots MU(5), MU(6),...,MU(NY-4) as well as the
additional knots MU(1) = MU(2) = MU(3) = MU(4) = Y(1) and MU
(NY-3) = MU(NY-2) = MU(NY-1) = MU(NY) = Y(MY) needed for the
B-spline representation.

14: C((NXEST-4)*(NYEST-4)) -- DOUBLE PRECISION array Output
On exit: the coefficients of the spline approximation. C(
(n-4)*(i-1)+j) is the coefficient c_{ij} defined in Section 3.

15: FP -- DOUBLE PRECISION Output
On exit: the sum of squared residuals, (theta), of the
computed spline approximation. If FP = 0.0, this is an
interpolating spline. FP should equal S within a relative
tolerance of 0.001 unless NX = NY = 8, when the spline has
no interior knots and so is simply a bicubic polynomial. For
knots to be inserted, S must be set to a value below the
value of FP produced in this case.

16: WRK(LWRK) -- DOUBLE PRECISION array Workspace
On entry: if the warm start option is used, the values WRK
(1),...,WRK(4) must be left unchanged from the previous call.
This array is used as workspace.

17: LWRK -- INTEGER Input
On entry:
the dimension of the array WRK as declared in the
(sub)program from which E02DCF is called.
Constraint:
\[
LWRK > 4 \times (M+M) + 11 \times (N+Y) + N \times M + \max(M,N) + 54.
\]

18: IWRK(LIWRK) -- INTEGER array Workspace
On entry: if the warm start option is used, the values IWRK
(1, ..., IWRK(3) must be left unchanged from the previous
call.

This array is used as workspace.

19: LIWRK -- INTEGER Input
On entry:
the dimension of the array IWRK as declared in the
(sub)program from which E02DCF is called.
Constraint: \[LIWRK \geq 3 + M + M + N + N\]

20: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry START /= 'C' or 'W',
or \[M < 4,\]
or \[M < 4,\]
or \[S < 0.0,\]
or \[S = 0.0 and N < M + 4,\]
or \( S = 0.0 \) and \( NYEST < MY + 4 \),

or \( NXEST < 8 \),

or \( NYEST < 8 \),

or \( LWRK < 4*(MX+MY)+11*(NXEST+NYEST)+NXEST*MY+\max(MY,NXEST)+54 \)

or \( LIWRK < 3 + MX + MY + NXEST + NYEST \).

**IFAIL = 2**

The values of \( X(q) \), for \( q = 1,2,\ldots, MX \), are not in strictly increasing order.

**IFAIL = 3**

The values of \( Y(r) \), for \( r = 1,2,\ldots, MY \), are not in strictly increasing order.

**IFAIL = 4**

The number of knots required is greater than allowed by \( NXEST \) and \( NYEST \). Try increasing \( NXEST \) and/or \( NYEST \) and, if necessary, supplying larger arrays for the parameters \( \text{LAMDA}, \text{MU}, \text{C}, \text{WRK} \) and \( \text{IWRK} \). However, if \( NXEST \) and \( NYEST \) are already large, say \( NXEST > MX/2 \) and \( NYEST > MY/2 \), then this error exit may indicate that \( S \) is too small.

**IFAIL = 5**

The iterative process used to compute the coefficients of the approximating spline has failed to converge. This error exit may occur if \( S \) has been set very small. If the error persists with increased \( S \), consult NAG.

If \( IFAIL = 4 \) or \( 5 \), a spline approximation is returned, but it fails to satisfy the fitting criterion (see (2) and (3) in Section 3) -- perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its sum of squared residuals \( FP \) is equal to the smoothing factor \( S \), up to a specified relative tolerance of 0.001 -- except that if \( n = 8 \) and \( n = 8 \), \( FP \) may be significantly less than \( S \); in this case the computed spline is simply the least-squares bicubic polynomial approximation of degree 3, i.e., a spline with no interior knots.

8. Further Comments
8.1. Timing

The time taken for a call of E02DCF depends on the complexity of the shape of the data, the value of the smoothing factor $S$, and the number of data points. If E02DCF is to be called for different values of $S$, much time can be saved by setting $\text{START} =$.

8.2. Weighting of Data Points

E02DCF does not allow individual weighting of the data values. If these were determined to widely differing accuracies, it may be better to use E02DDF. The computation time would be very much longer, however.

8.3. Choice of $S$

If the standard deviation of $f$ is the same for all $q$ and $r$

$$q,r$$

(the case for which this routine is designed - see Section 8.2.) and known to be equal, at least approximately, to $(\sigma)$, say, then following Reinsch [5] and choosing the smoothing factor $S$ in the range $(\sigma) (m+\sqrt{2}m)$, where $m=m_{x,y}$, is likely to give a good start in the search for a satisfactory value. If the standard deviations vary, the sum of their squares over all the data points could be used. Otherwise experimenting with different values of $S$ will be required from the start, taking account of the remarks in Section 3.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for $S$ and so determine the least-squares bicubic polynomial; the value returned for $FP$, call it $FP_0$, gives an upper bound for $S_0$.

Then progressively decrease the value of $S$ to obtain closer fits - say by a factor of 10 in the beginning, i.e., $S=FP_0/10$,

$$S=FP_0/100,$$ and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location, generally depend on the value of $S$ and on the behaviour of the function underlying the data. However, if E02DCF is called with $\text{START} = 'W'$, the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of $S$ and $\text{START} = 'W'$, a fit can finally be accepted as satisfactory, it may be worthwhile to call E02DCF once more with the selected value for $S$ but now using
START = 'C'. Often, E02DCF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.4. Choice of NXEST and NYEST

The number of knots may also depend on the upper bounds NXEST and NYEST. Indeed, if at a certain stage in E02DCF the number of knots in one direction (say n) has reached the value of its upper bound (NXEST), then from that moment on all subsequent knots are added in the other (y) direction. Therefore the user has the option of limiting the number of knots the routine locates in any direction. For example, by setting NXEST = 8 (the lowest allowable value for NXEST), the user can indicate that he wants an approximation which is a simple cubic polynomial in the variable x.

8.5. Outline of Method Used

If S=0, the requisite number of knots is known in advance, i.e., n = m +4 and n = m +4; the interior knots are located immediately as \( \lambda_i = x \) and \( \mu_j = y \), for \( i=5,6,...,n-4 \) and \( j=5,6,...,n-4 \). The corresponding least-squares spline is then an interpolating spline and therefore a solution of the problem.

If S>0, suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares, and (theta), the sum of squares of residuals, is computed. If (theta)>S, new knots are added to one knot set or the other so as to reduce (theta) at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of S and on the progress made so far in reducing (theta). Sooner or later, we find that (theta)<=S and at that point the knot sets are accepted. The routine then goes on to compute the (unique) spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has (theta)=S. The routine computes the spline by an iterative scheme which is ended when (theta)=S within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02BAF for least-squares curve fitting.

An exception occurs when the routine finds at the start that,
even with no interior knots \((n = n = 8)\), the least-squares spline already has its sum of residuals \(<= S\). In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure \((\eta)\), namely zero, it is returned at once as the (trivial) solution. It will usually mean that \(S\) has been chosen too large.

For further details of the algorithm and its use see Dierckx \([2]\).

8.6. Evaluation of Computed Spline

The values of the computed spline at the points \((TX(r), TY(r))\), for \(r = 1, 2, \ldots, N\), may be obtained in the double precision array \(FF\), of length at least \(N\), by the following code:

\[
\text{IFAIL} = 0 \\
\text{CALL E02DEF}(N, N, N, TX, TY, LAMDA, MU, C, FF, WRK, IWRK, IFAIL)
\]

where \(N\), \(N\), \(LAMDA\), \(MU\) and \(C\) are the output parameters of \(E02DCF\), \(WRK\) is a double precision workspace array of length at least \(NY-4\), and \(IWRK\) is an integer workspace array of length at least \(NY-4\).

To evaluate the computed spline on a \(KX\) by \(KY\) rectangular grid of points in the \(x\)-\(y\) plane, which is defined by the \(x\) co-ordinates stored in \(TX(q)\), for \(q = 1, 2, \ldots, KX\), and the \(y\) co-ordinates stored in \(TY(r)\), for \(r = 1, 2, \ldots, KY\), returning the results in the double precision array \(FG\) which is of length at least \(KX*KY\), the following call may be used:

\[
\text{IFAIL} = 0 \\
\text{CALL E02DFF}(KX, KY, NX, NY, TX, TY, LAMDA, MU, C, FG, WRK, LWRK, \\
* \text{IWRK, LIWRK, IFAIL})
\]

where \(NX\), \(NY\), \(LAMDA\), \(MU\) and \(C\) are the output parameters of \(E02DCF\), \(WRK\) is a double precision workspace array of length at least \(LWRK = \min(NWRK1, NWRK2)\), \(NWRK1 = KX*4+NX\), \(NWRK2 = KY*4+NY\), and \(IWRK\) is an integer workspace array of length at least \(LIWRK = KY + NY - 4\) if \(NWRK1 >= NWRK2\), or \(KX + NX - 4\) otherwise. The result of the spline evaluated at grid point \((q, r)\) is returned in element \((KY*(q-1)+r)\) of the array \(FG\).

9. Example

This example program reads in values of \(MX\), \(MY\), \(x\), for \(q = 1, 2, \ldots, q\), \(r\) ordinates \(f_{q,r}\) defined at the grid points \((x_q, y_r)\). It then calls
E02DCF to compute a bicubic spline approximation for one
specified value of S, and prints the values of the computed knots
and B-spline coefficients. Finally it evaluates the spline at a
small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

E02DDF -- Curve and Surface Fitting

E02DDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DDF computes a bicubic spline approximation to a set of
scattered data. The knots of the spline are located
automatically, but a single parameter must be specified to
control the trade-off between closeness of fit and smoothness of
fit.

2. Specification

SUBROUTINE E02DDF (START, M, X, Y, F, W, S, NXEST, NYEST,
1 NX, LAMDA, NY, MU, C, FP, RANK, WRK,
2 LWRK, IWRK, LIWRK, IFAIL)

INTEGER M, NXEST, NYEST, NX, NY, RANK, LWRK, IWRK
1 (LIWRK), LIWRK, IFAIL

DOUBLE PRECISION X(M), Y(M), F(M), W(M), S, LAMDA(NXEST),
1 MU(NYEST), C((NXEST-4)*(NYEST-4)), FP, WRK
2 (LWRK)

CHARACTER*1 START

3. Description

This routine determines a smooth bicubic spline approximation
s(x,y) to the set of data points (x ,y ,f ) with weights w , for
r r r r
r=1,2,...,m.

The approximation domain is considered to be the rectangle
[x ,x ]*[y ,y ], where x (y ) and x (y ) denote
min max min max min min max max
the lowest and highest data values of x (y ).
The spline is given in the B-spline representation

\[ s(x,y) = \sum_{i=1}^{n-4} \sum_{j=1}^{n-4} c_{ij} M_i(x) N_j(y), \quad (1) \]

where \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the former defined on the knots \((\lambda_i)\) to \((\lambda_{i+4})\) and the latter on the knots \((\mu_j)\) to \((\mu_{j+4})\). For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [1] for normalised B-splines.

The total numbers \( n \) and \( n \) of these knots and their values \( (\lambda_1), \ldots, (\lambda_n) \) and \( (\mu_1), \ldots, (\mu_n) \) are chosen automatically by the routine. The knots \((\lambda_1), \ldots, (\lambda_5)\) and \((\mu_1), \ldots, (\mu_5)\) are the interior knots; they divide the approximation domain \([x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]\) into \((n-7) \times (n-7)\) subpanels \([(\lambda_i), (\lambda_{i+1})] \times [(\mu_j), (\mu_{j+1})]\), for \( i=4,5,\ldots,n-4; j=4,5,\ldots,n-4 \). Then, much as in the curve case (see E02BEF), the coefficients \( c_{ij} \) are determined as the solution of the following constrained minimization problem:

\[ \text{minimize} \quad (\eta), \quad (2) \]

subject to the constraint

\[ \text{for } r=1 \]

\[ \text{minimize} \quad (\theta) = \sum_{r=1}^{m} 2 \]

\[ (\theta) = (\epsilon) \leq S \]
where: \( \eta \) is a measure of the (lack of) smoothness of \( s(x,y) \).
Its value depends on the discontinuity jumps in
\( s(x,y) \) across the boundaries of the subpanels. It is
zero only when there are no discontinuities and is
positive otherwise, increasing with the size of the
jumps (see Dierckx [2] for details).

\( \epsilon \) denotes the weighted residual \( w (f - s(x,y)) \),
and \( S \) is a non-negative number to be specified by the user.

By means of the parameter \( S \), 'the smoothing factor', the user
will then control the balance between smoothness and closeness of
fit, as measured by the sum of squares of residuals in (3). If \( S \)
is too large, the spline will be too smooth and signal will be
lost (underfit); if \( S \) is too small, the spline will pick up too
much noise (overfit). In the extreme cases the method would
return an interpolating spline \( ((\theta) = 0) \) if \( S \) were set to zero,
and returns the least-squares bicubic polynomial \( ((\eta) = 0) \) if \( S \)
is set very large. Experimenting with \( S \)-values between these two
extremes should result in a good compromise. (See Section 8.2 for
advice on choice of \( S \).) Note however, that this routine, unlike
E02BEF and E02DCF, does not allow \( S \) to be set exactly to zero: to
compute an interpolant to scattered data, E01SAF or E01SEF should
be used.

The method employed is outlined in Section 8.5 and fully
described in Dierckx [2] and [3]. It involves an adaptive
strategy for locating the knots of the bicubic spline (depending
on the function underlying the data and on the value of \( S \)), and
an iterative method for solving the constrained minimization
problem once the knots have been determined.

Values of the computed spline can subsequently be computed by
calling E02DEF or E02DFF as described in Section 8.6.

4. References

Theory. 6 50--62.


with Spline Functions. Report TW54. Department of Computer
Science, Katholieke Universiteit Leuven.
5. Parameters

1: START -- CHARACTER*1
On entry: START must be set to 'C' or 'W'.

If START = 'C' (Cold start), the routine will build up the
knot set starting with no interior knots. No values need be
assigned to the parameters NX, NY, LAMDA, MU or WRK.

If START = 'W' (Warm start), the routine will restart the
knot-placing strategy using the knots found in a previous
call of the routine. In this case, the parameters NX, NY,
LAMDA, MU and WRK must be unchanged from that previous
call. This warm start can save much time in searching for a
satisfactory value of S. Constraint: START = 'C' or 'W'.

2: M -- INTEGER
On entry: m, the number of data points.

The number of data points with non-zero weight (see W below)
must be at least 16.

3: X(M) -- DOUBLE PRECISION array
Input

4: Y(M) -- DOUBLE PRECISION array
Input

5: F(M) -- DOUBLE PRECISION array
Input
On entry: X(r), Y(r), F(r) must be set to the co-ordinates
of (x , y , f ), the rth data point, for r=1,2,...,m. The
r r r
order of the data points is immaterial.

6: W(M) -- DOUBLE PRECISION array
Input
On entry: W(r) must be set to w , the rth value in the set
r
of weights, for r=1,2,...,m. Zero weights are permitted and
the corresponding points are ignored, except when
determining x , x , y and y (see Section 8.4). For
min max min max
advice on the choice of weights, see Section 2.1.2 of the
Chapter Introduction. Constraint: the number of data points with non-zero weight must be at least 16.

7: S -- DOUBLE PRECISION
    On entry: the smoothing factor, S.
    For advice on the choice of S, see Section 3 and Section 8.2. Constraint: S > 0.0.

8: NXEST -- INTEGER
    Input

9: NYEST -- INTEGER
    Input
    On entry: an upper bound for the number of knots \(n_x\) and \(n_y\)
    required in the x- and y-directions respectively.
    In most practical situations, \(NXEST = NYEST = 4 + \sqrt{m}/2\) is sufficient. See also Section 8.3. Constraint: \(NXEST \geq 8\) and \(NYEST \geq 8\).

10: NX -- INTEGER
    Input/Output
    On entry: if the warm start option is used, the value of \(NX\)
    must be left unchanged from the previous call. On exit: the
    total number of knots, \(n_x\), of the computed spline with
    respect to the x variable.

11: LAMDA(NXEST) -- DOUBLE PRECISION array
    Input/Output
    On entry: if the warm start option is used, the values LAMDA(1), LAMDA(2),...,LAMDA(NX) must be left unchanged from the previous call. On exit: LAMDA contains the complete set of knots (lambda) associated with the x variable, i.e., the interior knots LAMDA(5), LAMDA(6),...,LAMDA(NX-4) as well as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = \(x\) and LAMDA(NX-3) = LAMDA(NX-2) = LAMDA(NX-1) = \(\min\) LAMDA(NX) = \(x\) needed for the B-spline representation max (where \(x\) and \(x\) are as described in Section 3). min max

12: NY -- INTEGER
    Input/Output
    On entry: if the warm start option is used, the value of \(NY\)
    must be left unchanged from the previous call. On exit: the
    total number of knots, \(n_y\), of the computed spline with
    respect to the y variable.
13: MU(NYEST) -- DOUBLE PRECISION array Input/Output
   On entry: if the warm start option is used, the values MU(1),
   MU(2),...,MU(NY) must be left unchanged from the previous
   call. On exit: MU contains the complete set of knots (mu)_i
   associated with the y variable, i.e., the interior knots MU
   (5), MU(6),...,MU(NY-4) as well as the additional knots MU
   (1) = MU(2) = MU(3) = MU(4) = y_max and MU(NY-3) = MU(NY-2) =
   min
   MU(NY-1) = MU(NY) = y_min needed for the B-spline
   representation (where y_min and y_max are as described in
   Section 3).

14: C((NXEST-4)*(NYEST-4)) -- DOUBLE PRECISION array Output
   On exit: the coefficients of the spline approximation. C((
   n-4)*(i-1)+j) is the coefficient c_yij defined in Section 3.

15: FP -- DOUBLE PRECISION Output
   On exit: the weighted sum of squared residuals, (theta), of
   the computed spline approximation. FP should equal S within
   a relative tolerance of 0.001 unless NX = NY = 8, when the
   spline has no interior knots and so is simply a bicubic
   polynomial. For knots to be inserted, S must be set to a
   value below the value of FP produced in this case.

16: RANK -- INTEGER Output
   On exit: RANK gives the rank of the system of equations used
   to compute the final spline (as determined by a suitable
   machine-dependent threshold). When RANK = (NX-4)*(NY-4), the
   solution is unique; otherwise the system is rank-deficient
   and the minimum-norm solution is computed. The latter case
   may be caused by too small a value of S.

17: WRK(LWRK) -- DOUBLE PRECISION array Workspace
   On entry: if the warm start option is used, the value of WRK
   (1) must be left unchanged from the previous call.
   This array is used as workspace.

18: LWRK -- INTEGER Input
   On entry:
   the dimension of the array WRK as declared in the
   (sub)program from which E02DDF is called.
   Constraint: LWRK >= (7*u*v+25*w)*(w+1)+2*(u+v+4*M)+23*w+56,
   where
For some problems, the routine may need to compute the minimal least-squares solution of a rank-deficient system of linear equations (see Section 3). The amount of workspace required to solve such problems will be larger than specified by the value given above, which must be increased by an amount, LWRK2 say. An upper bound for LWRK2 is given by $4u*v*w+2u*v+4*w$, where $u$, $v$ and $w$ are as above. However, if there are enough data points, scattered uniformly over the approximation domain, and if the smoothing factor $S$ is not too small, there is a good chance that this extra workspace is not needed. A lot of memory might therefore be saved by assuming $LWRK2 = 0$.

19: IWRK(LIWRK) -- INTEGER array

Workspace

20: LIWRK -- INTEGER

Input

On entry:
the dimension of the array IWRK as declared in the
(sub)program from which E02DDF is called.
Constraint: $LIWRK \geq M + 2 \times (NXEST-7) \times (NYEST-7)$.

21: IFAIL -- INTEGER

Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry $IFAIL = 0$ or $-1$, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

$IFAIL= 1$
On entry $START /= 'C' \text{ or } 'W'$,

or the number of data points with non-zero weight < 16,

or $S <= 0.0$,

or $NXEST < 8$,

or $NYEST < 8$,
or \[ LWRK < (7u^2v+25w)(w+1)+2(u+v+4M)+23w+56, \]
where \( u = NXEST - 4, \) \( v = NYEST - 4 \) and \( w = \max(u,v), \)

or \[ LIWRK < M+2(NXEST-7)(NYEST-7). \]

IFAIL = 2
On entry either all the \( X(r), r = 1,2,\ldots,M, \) are equal,
or all the \( Y(r), r = 1,2,\ldots,M, \) are equal.

IFAIL = 3
The number of knots required is greater than allowed by
\( NXEST \) and \( NYEST. \) Try increasing \( NXEST \) and/or \( NYEST \) and, if
necessary, supplying larger arrays for the parameters \( LAMDA, \)
\( MU, C, WRK \) and \( IWRK. \) However, if \( NXEST \) and \( NYEST \) are already
large, say \( NXEST, NYEST > 4 + \sqrt{M}/2, \) then this error exit
may indicate that \( S \) is too small.

IFAIL = 4
No more knots can be added because the number of B-spline
coefficients \((NX-4)*(NY-4)\) already exceeds the number of
data points \( M. \) This error exit may occur if either of \( S \) or \( M \)
is too small.

IFAIL = 5
No more knots can be added because the additional knot would
(quasi) coincide with an old one. This error exit may occur
if too large a weight has been given to an inaccurate data
point, or if \( S \) is too small.

IFAIL = 6
The iterative process used to compute the coefficients of
the approximating spline has failed to converge. This error
exit may occur if \( S \) has been set very small. If the error
persists with increased \( S, \) consult NAG.

IFAIL = 7
\( LWRK \) is too small; the routine needs to compute the minimal
least-squares solution of a rank-deficient system of linear
equations, but there is not enough workspace. There is no
approximation returned but, having saved the information
contained in \( NX, LAMDA, NY, MU \) and \( WRK, \) and having adjusted
the value of \( LWRK \) and the dimension of array \( WRK \)
accordingly, the user can continue at the point the program
was left by calling \texttt{E02DDF} with \texttt{START} = \texttt{'W'}. Note that the
requested value for \( LWRK \) is only large enough for the
current phase of the algorithm. If the routine is restarted
with \( LWRK \) set to the minimum value requested, a larger
request may be made at a later stage of the computation. See Section 5 for the upper bound on LWRK. On soft failure, the minimum requested value for LWRK is returned in IWRK(1) and the safe value for LWRK is returned in IWRK(2).

If IFAIL = 3, 4, 5 or 6, a spline approximation is returned, but it fails to satisfy the fitting criterion (see (2) and (3) in Section 3 -- perhaps only by a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its weighted sum of squared residuals FP is equal to the smoothing factor S, up to a specified relative tolerance of 0.001 - except that if n = 8 and n = 8, FP may be significantly less than S: in this case the computed spline is simply the least-squares bicubic polynomial approximation of degree 3, i.e., a spline with no interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02DDF depends on the complexity of the shape of the data, the value of the smoothing factor S, and the number of data points. If E02DDF is to be called for different values of S, much time can be saved by setting START = It should be noted that choosing S very small considerably increases computation time.

8.2. Choice of S

If the weights have been correctly chosen (see Section 2.1.2 of the Chapter Introduction), the standard deviation of w f would be the same for all r, equal to (sigma), say. In this case, choosing the smoothing factor S in the range (sigma) (m+-\sqrt{2m}), as suggested by Reinsch [6], is likely to give a good start in the search for a satisfactory value. Otherwise, experimenting with different values of S will be required from the start.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for S and so determine the least-squares bicubic polynomial; the value returned for FP, call it FP, gives an upper bound for S.

Then progressively decrease the value of S to obtain closer fits - say by a factor of 10 in the beginning, i.e., S = FP / 10,
S = FP /100, and so on, and more carefully as the approximation shows more details.

To choose S very small is strongly discouraged. This considerably increases computation time and memory requirements. It may also cause rank-deficiency (as indicated by the parameter RANK) and endanger numerical stability.

The number of knots of the spline returned, and their location, generally depend on the value of S and on the behaviour of the function underlying the data. However, if E02DDF is called with START = 'W', the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of S and START = 'W', a fit can finally be accepted as satisfactory, it may be worthwhile to call E02DDF once more with the selected value for S but now using START = 'C'. Often, E02DDF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Choice of NXEST and NYEST

The number of knots may also depend on the upper bounds NXEST and NYEST. Indeed, if at a certain stage in E02DDF the number of knots in one direction (say n) has reached the value of its upper bound (NXEST), then from that moment on all subsequent knots are added in the other (y) direction. This may indicate that the value of NXEST is too small. On the other hand, it gives the user the option of limiting the number of knots the routine locates in any direction. For example, by setting NXEST = 8 (the lowest allowable value for NXEST), the user can indicate that he wants an approximation which is a simple cubic polynomial in the variable x.

8.4. Restriction of the approximation domain

The fit obtained is not defined outside the rectangle \([((\lambda), (\lambda), \mu, \mu)]\). The reason for taking \(n - 3\) the extreme data values of x and y for these four knots is that, as is usual in data fitting, the fit cannot be expected to give satisfactory values outside the data region. If, nevertheless, the user requires values over a larger rectangle, this can be achieved by augmenting the data with two artificial data points \((a, c, 0)\) and \((b, d, 0)\) with zero weight, where \([a, b]\)[c, d] denotes the enlarged rectangle.
8.5. Outline of method used

First suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares and (theta), the sum of squares of residuals, is computed. If (theta)>S, a new knot is added to one knot set or the other so as to reduce (theta) at the next stage. The new knot is located in an interval where the fit is particularly poor. Sooner or later, we find that (theta)<=S and at that point the knot sets are accepted. The routine then goes on to compute a spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has (theta)=S.

An exception occurs when the routine finds at the start that, even with no interior knots (N = 8), the least-squares spline already has its sum of squares of residuals <=S. In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure (eta), namely zero, it is returned at once as the (trivial) solution. It will usually mean that S has been chosen too large.

For further details of the algorithm and its use see Dierckx [2].

8.6. Evaluation of computed spline

The values of the computed spline at the points (TX(r),TY(r)), for r = 1, 2,...,N, may be obtained in the double precision array FF, of length at least N, by the following code:

```fortran
IFAIL = 0
CALL E02DEF(N,NX,NY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DDF, WRK is a double precision workspace array of length at least NY-4, and IWRK is an integer workspace array of length at least NY-4.

To evaluate the computed spline on a KX by KY rectangular grid of
points in the x-y plane, which is defined by the x co-ordinates stored in TX(q), for q=1,2,...,KX, and the y co-ordinates stored in TY(r), for r=1,2,...,KY, returning the results in the double precision array FG which is of length at least KX*KY, the following call may be used:

```fortran
IFAIL = 0
CALL E02DFF(KX,KY,NX,NY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK, *
              IWRK,LIWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DDF, WRK is a double precision workspace array of length at least LWRK = min(NWRK1,NWRK2), NWRK1 = KX*4+NX, NWRK2 = KY*4+NY, and IWRK is an integer workspace array of length at least LIWRK = KY + NY - 4 if NWRK1 >= NWRK2, or KX + NX - 4 otherwise. The result of the spline evaluated at grid point (q,r) is returned in element (KY*(q-1)+r) of the array FG.

9. Example

This example program reads in a value of M, followed by a set of M data points (x ,y ,f ) and their weights w . It then calls E02DDF to compute a bicubic spline approximation for one specified value of S, and prints the values of the computed knots and B-spline coefficients. Finally it evaluates the spline at a small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
SUBROUTINE E02DEF (M, PX, PY, X, Y, LAMDA, MU, C, FF, WRK, 
1          IWRK, IFAIL)
INTEGER M, PX, PY, IWRK(PY-4), IFAIL
DOUBLE PRECISION X(M), Y(M), LAMDA(PX), MU(PY), C((PX-4)* 
1          (PY-4)), FF(M), WRK(PY-4)

3. Description

This routine calculates values of the bicubic spline $s(x,y)$ at 
prescribed points $(x_r, y_r)$, for $r=1,2,...,m$, from its augmented 
knot sets $\{(\lambda_i)\}$ and $\{(\mu_j)\}$ and from the coefficients $c_{ij}$, 
for $i=1,2,...,PX-4; j=1,2,...,PY-4$, in its B-spline 
representation

$$
--
\begin{align*}
  s(x,y) &= c_{ij} M_i(x) N_j(y), \\
         & \quad \text{for } i=1,2,...,PX-4; j=1,2,...,PY-4.
\end{align*}
$$

Here $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines, the 
i former defined on the knots $(\lambda_i)$ to $(\lambda_{i+4})$ and the 
j latter on the knots $(\mu_j)$ to $(\mu_{j+4})$.

This routine may be used to calculate values of a bicubic spline 
given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF. 
It is derived from the routine B2VRE in Anthony et al [1].

4. References

Approximation Subroutine Library. National Physical 
Laboratory.


5. Parameters

1: M -- INTEGER                        Input
   On entry: $m$, the number of points at which values of the 
spline are required. Constraint: $M \geq 1$.

2: PX -- INTEGER                        Input
3: PY -- INTEGER Input
On entry: PX and PY must specify the total number of knots
associated with the variables x and y respectively. They are
such that PX-8 and PY-8 are the corresponding numbers of
interior knots. Constraint: PX >= 8 and PY >= 8.

4: X(M) -- DOUBLE PRECISION array Input
5: Y(M) -- DOUBLE PRECISION array Input
On entry: X and Y must contain x and y , for r=1,2,...,m,
r r
respectively. These are the co-ordinates of the points at
which values of the spline are required. The order of the
points is immaterial. Constraint: X and Y must satisfy

LAMDA(4) <= X(r) <= LAMDA(PX-3)

and

MU(4) <= Y(r) <= MU(PY-3), for r=1,2,...,m.

The spline representation is not valid outside these
intervals.

6: LAMDA(PX) -- DOUBLE PRECISION array Input
7: MU(PY) -- DOUBLE PRECISION array Input
On entry: LAMDA and MU must contain the complete sets of
knots {(lambda)} and {(mu)} associated with the x and y
variables respectively. Constraint: the knots in each set
must be in non-decreasing order, with LAMDA(PX-3) > LAMDA(4)
and MU(PY-3) > MU(4).

8: C((PX-4)*(PY-4)) -- DOUBLE PRECISION array Input
On entry: C((PX-4)*(i-1)+j) must contain the coefficient
c described in Section 3, for i=1,2,...,PX-4;
ij
j=1,2,...,PY-4.

9: FF(M) -- DOUBLE PRECISION array Output
On exit: FF(r) contains the value of the spline at the
point (x ,y ), for r=1,2,...,m.
r r

10: WRK(PY-4) -- DOUBLE PRECISION array Workspace
11: IWRK(PY-4) -- INTEGER array Workspace
12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry M < 1,

or PY < 8,

or PX < 8.

IFAIL = 2
On entry the knots in array LAMDA, or those in array MU, are
not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or
MU(PY-3) <= MU(4).

IFAIL = 3
On entry at least one of the prescribed points (x ,y ) lies
r r
outside the rectangle defined by LAMDA(4), LAMDA(PX-3) and
MU(4), MU(PY-3).

7. Accuracy

The method used to evaluate the B-splines is numerically stable,
in the sense that each computed value of s(x ,y ) can be regarded
r r
as the value that would have been obtained in exact arithmetic
from slightly perturbed B-spline coefficients. See Cox [2] for
details.

8. Further Comments

Computation time is approximately proportional to the number of
points, m, at which the evaluation is required.

9. Example

This program reads in knot sets LAMDA(1), ..., LAMDA(PX) and MU(1)
,..., MU(PY), and a set of bicubic spline coefficients c .
Following these are a value for \( m \) and the co-ordinates \((x_r, y_r)\), for \( r=1,2,...,m \), at which the spline is to be evaluated.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**E02 -- Curve and Surface Fitting**

E02DFF  -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02DFF calculates values of a bicubic spline from its B-spline representation. The spline is evaluated at all points on a rectangular grid.

2. Specification

```fortran
SUBROUTINE E02DFF (MX, MY, PX, PY, X, Y, LAMDA, MU, C, FF,
  1 WRK, LWRK, IWRK(LIWRK), LIWRK, IFAIL)
  INTEGER MX, MY, PX, PY, LWRK, IWRK(LIWRK), LIWRK,
  1 IFAIL
  DOUBLE PRECISION X(MX), Y(MY), LAMDA(PX), MU(PY), C((PX-4)*
  1 (PY-4)), FF(MX*MY), WRK(LWRK)
```

3. Description

This routine calculates values of the bicubic spline \( s(x,y) \) on a rectangular grid of points in the \( x-y \) plane, from its augmented knot sets \{\( \lambda \}\} and \{\( \mu \}\} and from the coefficients \( c_{ij} \) for \( i=1,2,...,PX-4; j=1,2,...,PY-4 \), in its B-spline representation

\[
--
\]

\[
 s(x,y) = \sum_{i=1}^{PX-4} \sum_{j=1}^{PY-4} c_{ij} M_i(x) N_j(y).

-- ij
\]

Here \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the former defined on the knots \( \lambda \) to \( \lambda \) and the
latter on the knots (μi) to (μi+4).

The points in the grid are defined by co-ordinates xq for q = 1, 2, ..., m, along the x axis, and co-ordinates yr for r = 1, 2, ..., m along the y axis.

This routine may be used to calculate values of a bicubic spline given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF. It is derived from the routine B2VRE in Anthony et al [1].

4. References


5. Parameters

1: MX -- INTEGER Input

2: MY -- INTEGER Input

On entry: MX and MY must specify m and m respectively, the number of points along the x and y axis that define the rectangular grid. Constraint: MX >= 1 and MY >= 1.

3: PX -- INTEGER Input

4: PY -- INTEGER Input

On entry: PX and PY must specify the total number of knots associated with the variables x and y respectively. They are such that PX-8 and PY-8 are the corresponding numbers of interior knots. Constraint: PX >= 8 and PY >= 8.

5: X(MX) -- DOUBLE PRECISION array Input

6: Y(MY) -- DOUBLE PRECISION array Input

On entry: X and Y must contain xq for q = 1, 2, ..., m, and yr for r = 1, 2, ..., m, respectively. These are the x and y co-ordinates that define the rectangular grid of points at
which values of the spline are required. Constraint: X and Y must satisfy

\[ LAMDA(4) \leq X(q) < X(q+1) \leq LAMDA(PX-3), \text{ for } q=1,2,\ldots,m-1 \]

and

\[ MU(4) \leq Y(r) < Y(r+1) \leq MU(PY-3), \text{ for } r=1,2,\ldots,m-1. \]

The spline representation is not valid outside these intervals.

7: LAMDA(PX) -- DOUBLE PRECISION array
   Input

8: MU(PY) -- DOUBLE PRECISION array
   Input
   On entry: LAMDA and MU must contain the complete sets of
   knots \{(lambda)\} and \{(mu)\} associated with the x and y
   variables respectively. Constraint: the knots in each set
   must be in non-decreasing order, with LAMDA(PX-3) > LAMDA(4)
   and MU(PY-3) > MU(4).

9: C((PX-4)*(PY-4)) -- DOUBLE PRECISION array
   Input
   On entry: C((PY-4)*(i-1)+j) must contain the coefficient
   \(c\) described in Section 3, for \(i=1,2,\ldots,PX-4;\)
   \(j=1,2,\ldots,PY-4.\)

10: FF(MX*MY) -- DOUBLE PRECISION array
    Output
    On exit: FF(MY*(q-1)+r) contains the value of the spline at
    the point \((x,Y)\), for \(q=1,2,\ldots,m; r=1,2,\ldots,m\).

11: WRK(LWRK) -- DOUBLE PRECISION array
    Workspace

12: LWRK -- INTEGER
    Input
    On entry:
    the dimension of the array WRK as declared in the
    (sub)program from which E02DFF is called.
    Constraint: LWRK >= min(NWRK1,NWRK2), where NWRK1=4*MX+PX,
    NWRK2=4*MY+PY.

13: LIWRK -- INTEGER array
    Workspace

14: LIWRK -- INTEGER
    Input
    On entry:
    the dimension of the array IWRK as declared in the
    (sub)program from which E02DFF is called.
    Constraint: LIWRK >= MY + PY - 4 if NWRK1 > NWRK2, or MX +
    PX - 4 otherwise, where NWRK1 and NWRK2 are as defined in
the description of argument LWRK.

15: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry MX < 1,
or MY < 1,
or PY < 8,
or PX < 8.

IFAIL= 2
On entry LWRK is too small,
or LIWRK is too small.

IFAIL= 3
On entry the knots in array LAMDA, or those in array MU, are not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or MU(PY-3) <= MU(4).

IFAIL= 4
On entry the restriction LAMDA(4) <= X(1) < ... < X(MX) <= LAMDA(PX-3), or the restriction MU(4) <= Y(1) < ... < Y(MY) <= MU(PY-3), is violated.

7. Accuracy

The method used to evaluate the B-splines is numerically stable, in the sense that each computed value of $s(x, y)$ can be regarded as the value that would have been obtained in exact arithmetic from slightly perturbed B-spline coefficients. See Cox [2] for details.
8. Further Comments

Computation time is approximately proportional to m m +4(m +m )
   x y x y

9. Example

This program reads in knot sets LAMDA(1),..., LAMDA(PX) and MU(1)
   ,..., MU(PY), and a set of bicubic spline coefficients c .
   ij
Following these are values for m and the x co-ordinates x , for
   x q
q=1,2,...,m , and values for m and the y co-ordinates y , for
   x y r
r=1,2,...,m , defining the grid of points on which the spline is
   y to be evaluated.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting
E02GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02GAF calculates an l solution to an over-determined system of
   1
linear equations.

2. Specification

SUBROUTINE E02GAF (M, A, LA, B, NPLUS2, TOLER, X, RESID,
   1   IRANK, ITER, IWORK, IFAIL)
   INTEGER M, LA, NPLUS2, IRANK, ITER, IWORK(M),
   1   IFAIL
   DOUBLE PRECISION A(LA,NPLUS2), B(M), TOLER, X(NPLUS2),
   1   RESID

3. Description

Given a matrix A with m rows and n columns (m>=n) and a vector b
with \( m \) elements, the routine calculates an \( l_1 \) solution to the \( l_1 \) over-determined system of equations

\[
Ax=b.
\]

That is to say, it calculates a vector \( x \), with \( n \) elements, which minimizes the \( l_1 \)-norm (the sum of the absolute values) of the \( l_1 \) residuals

\[
r(x) = \sum_{i=1}^{m} |r_i|,
\]

where the residuals \( r \) are given by

\[
r_i = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad i=1,2,...,m.
\]

Here \( a_{ij} \) is the element in row \( i \) and column \( j \) of \( A \), \( b_i \) is the \( i \)th element of \( b \) and \( x_j \) the \( j \)th element of \( x \). The matrix \( A \) need not be of full rank.

Typically in applications to data fitting, data consisting of \( m \) points with co-ordinates \((t_i,y_i)\) are to be approximated in the \( l_1 \)-norm by a linear combination of known functions \((\phi_j)(t)\),

\[
(\alpha_1)(\phi_1)(t)+\alpha_2(\phi_2)(t)+...+\alpha_n(\phi_n)(t).
\]

This is equivalent to fitting an \( l_1 \) solution to the over-determined system of equations

\[
\sum_{j=1}^{n} (\phi_j)(t_i)(\alpha_j) = y_i, \quad i=1,2,...,m.
\]
Thus if, for each value of $i$ and $j$, the element $a_{ij}$ of the matrix $A$ in the previous paragraph is set equal to the value of $(\phi_i(t))$ and $b$ is set equal to $y$, the solution vector $x$ will contain the required values of the $(\alpha_j)$. Note that the independent variable $t$ above can, instead, be a vector of several independent variables (this includes the case where each $(\phi_i)$ is a function of a different variable, or set of variables).

The algorithm is a modification of the simplex method of linear programming applied to the primal formulation of the $l_1$ problem (see Barrodale and Roberts [1] and [2]). The modification allows several neighbouring simplex vertices to be passed through in a single iteration, providing a substantial improvement in efficiency.

4. References


5. Parameters

1: $M$ -- INTEGER
   On entry: the number of equations, $m$ (the number of rows of the matrix $A$). Constraint: $M \geq n \geq 1$.

2: $A(LA,NPLUS2)$ -- DOUBLE PRECISION array
   On entry: $A(i,j)$ must contain $a_{ij}$, the element in the $i$th row and $j$th column of the matrix $A$, for $i=1,2,\ldots,m$ and $j=1,2,\ldots,n$. The remaining elements need not be set. On exit: $A$ contains the last simplex tableau generated by the simplex method.

3: $LA$ -- INTEGER
   On entry: the first dimension of the array $A$ as declared in the
(sub)program from which E02GAF is called.
Constraint: LA >= M + 2.

4: B(M) -- DOUBLE PRECISION array
   Input/Output
   On entry: b, the ith element of the vector b, for
   
i=1,2,...,m. On exit: the ith residual r corresponding to
   
i the solution vector x, for i=1,2,...,m.

5: NPLUS2 -- INTEGER
   Input
   On entry: n+2, where n is the number of unknowns (the
   number of columns of the matrix A). Constraint: 3 <= NPLUS2
   <= M + 2.

6: TOLER -- DOUBLE PRECISION
   Input
   On entry: a non-negative value. In general TOLER specifies
   a threshold below which numbers are regarded as zero. The
   recommended threshold value is (epsilon) where (epsilon)
   is the machine precision. The recommended value can be
   computed within the routine by setting TOLER to zero. If
   premature termination occurs a larger value for TOLER may
   result in a valid solution. Suggested value: 0.0.

7: X(NPLUS2) -- DOUBLE PRECISION array
   Output
   On exit: X(j) contains the jth element of the solution
   vector x, for j=1,2,...,n. The elements X(n+1) and X(n+2)
   are unused.

8: RESID -- DOUBLE PRECISION
   Output
   On exit: the sum of the absolute values of the residuals
   for the solution vector x.

9: IRANK -- INTEGER
   Output
   On exit: the computed rank of the matrix A.

10: ITER -- INTEGER
    Output
    On exit: the number of iterations taken by the simplex
    method.

11: IWORK(M) -- INTEGER array
    Workspace

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
An optimal solution has been obtained but this may not be unique.

IFAIL= 2
The calculations have terminated prematurely due to rounding errors. Experiment with larger values of TOLER or try scaling the columns of the matrix (see Section 8).

IFAIL= 3
On entry NPLUS2 < 3,

or NPLUS2 > M + 2,

or LA < M + 2.

7. Accuracy

Experience suggests that the computational accuracy of the solution x is comparable with the accuracy that could be obtained by applying Gaussian elimination with partial pivoting to the n equations satisfied by this algorithm (i.e., those equations with zero residuals). The accuracy therefore varies with the conditioning of the problem, but has been found generally very satisfactory in practice.

8. Further Comments

The effects of m and n on the time and on the number of iterations in the Simplex Method vary from problem to problem, but typically the number of iterations is a small multiple of n and the total time taken by the routine is approximately

\[ \frac{2}{mm} \]

proportional to mn.

It is recommended that, before the routine is entered, the columns of the matrix A are scaled so that the largest element in each column is of the order of unity. This should improve the conditioning of the matrix, and also enable the parameter TOLER to perform its correct function. The solution x obtained will then, of course, relate to the scaled form of the matrix. Thus if the scaling is such that, for each j=1,2,...,n, the elements of the jth column are multiplied by the constant k, the element x

\[ x_j \]

of the solution vector x must be multiplied by k if it is
desired to recover the solution corresponding to the original matrix A.

9. Example

Suppose we wish to approximate a set of data by a curve of the form

\[ y = Ke + Le + M \]

where K, L and M are unknown. Given values \( y_i \) at 5 points \( t_i \) we may form the over-determined set of equations for K, L and M

\[ x_i - x_i \]
\[ e_i K + e_i L + M = y_i, \quad i = 1, 2, \ldots, 5. \]

E02GAF is used to solve these in the 1 sense.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting

E02ZAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02ZAF sorts two-dimensional data into rectangular panels.

2. Specification

```fortran
SUBROUTINE E02ZAF (PX, PY, LAMDA, MU, M, X, Y, POINT, NPOINT, ADRES, NADRES, IFAIL)
INTEGER PX, PY, M, POINT(NPOINT), NPOINT, ADRES, NADRES, IFAIL
DOUBLE PRECISION LAMDA(PX), MU(PY), X(M), Y(M)
```
3. Description

A set of m data points with rectangular Cartesian co-ordinates x, y are sorted into panels defined by lines parallel to the y and x axes. The intercepts of these lines on the x and y axes are given in LAMDA(i), for i=5,6,...,PX-4 and MU(j), for j=5,6,...,PY-4, respectively. The subroutine orders the data so that all points in a panel occur before data in succeeding panels, where the panels are numbered from bottom to top and then left to right, with the usual arrangement of axes, as shown in the diagram. Within a panel the points maintain their original order.

Please see figure in printed Reference Manual

A data point lying exactly on one or more panel sides is taken to be in the highest-numbered panel adjacent to the point. The subroutine does not physically rearrange the data, but provides the array POINT which contains a linked list for each panel, pointing to the data in that panel. The total number of panels is (PX-7)*(PY-7).

4. References

None.

5. Parameters

1: PX -- INTEGER Input

2: PY -- INTEGER Input

On entry: PX and PY must specify eight more than the number of intercepts on the x axis and y axis, respectively. Constraint: PX >= 8 and PY >= 8.

3: LAMDA(PX) -- DOUBLE PRECISION array Input

On entry: LAMDA(5) to LAMDA(PX-4) must contain, in non-decreasing order, the intercepts on the x axis of the sides of the panels parallel to the y axis.

4: MU(PY) -- DOUBLE PRECISION array Input

On entry: MU(5) to MU(PY-4) must contain, in non-decreasing order, the intercepts on the y axis of the sides of the panels parallel to the x axis.

5: M -- INTEGER Input

On entry: the number m of data points.
6: \( X(M) \) -- DOUBLE PRECISION array 
   \( M \) \( \times \) \( M \) \( \times \) \( M \) \( \times \) \( M \) 
   Input

7: \( Y(M) \) -- DOUBLE PRECISION array 
   \( M \) \( \times \) \( M \) \( \times \) \( M \) \( \times \) \( M \) 
   On entry: the co-ordinates of the \( r \)th data point \((x_r, y_r)\), 
   for \( r = 1, 2, \ldots, m \).

8: \( \text{POINT}(NPOINT) \) -- INTEGER array 
   Output 
   On exit: for \( i = 1, 2, \ldots, NADRES \), \( \text{POINT}(m+i) = I_1 \) is the 
   index of the first point in panel \( i \), \( \text{POINT}(I_1) = I_2 \) is the 
   index of the second point in panel \( i \) and so on. 

\( \text{POINT}(IN) = 0 \) indicates that \( X(IN), Y(IN) \) was the last point 
in the panel.

The co-ordinates of points in panel \( i \) can be accessed in 
turn by means of the following instructions:
   \( IN = M + I \)
   10 \( IN = \text{POINT}(IN) \)
   IF (IN.EQ. 0) GOTO 20
   \( XI = X(IN) \)
   \( YI = Y(IN) \)
   .
   .
   GOTO 10
20...

9: \( \text{NPOINT} \) -- INTEGER 
   Input 
   On entry:
   the dimension of the array \( \text{POINT} \) as declared in the 
   (sub)program from which \( \text{E02ZAF} \) is called. 
   Constraint: \( \text{NPOINT} \geq M + (PX-7)*(PY-7) \).

10: \( \text{ADRES}(NADRES) \) -- INTEGER array 
    Workspace

11: \( NADRES \) -- INTEGER 
    Input 
    On entry: the value \( (PX-7)*(PY-7) \), the number of panels 
    into which the \((x,y)\) plane is divided.

12: \( IFAIL \) -- INTEGER 
    Input/Output 
    On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not 
    familiar with this parameter (described in the Essential 
    Introduction) the recommended value is 0. 
    On exit: \( IFAIL = 0 \) unless the routine detects an error (see 
    Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   The intercepts in the array LAMDA, or in the array MU, are not in non-decreasing order.

IFAIL = 2
   On entry PX < 8,
   or PY < 8,
   or M <= 0,
   or NADRES /= (PX-7)*(PY-7),
   or NPOINT < M + (PX-7)*(PY-7).

7. Accuracy

Not applicable.

8. Further Comments

The time taken by this routine is approximately proportional to m*log(NADRES).

This subroutine was written to sort two dimensional data in the manner required by routines E02DAF and E02DBF(*). The first 9 parameters of E02ZAF are the same as the parameters in E02DAF and E02DBF(*) which have the same name.

9. Example

This example program reads in data points and the intercepts of the panel sides on the x and y axes; it calls E02ZAF to set up the index array POINT; and finally it prints the data points in panel order.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagFittingPackage (NAGE02)

Exports:

| package NAGE02 NagFittingPackage |

)abbrev package NAGE02 NagFittingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:59 1994
++ Description:
++ This package uses the NAG Library to find a
++ function which approximates a set of data points. Typically the
++ data contain random errors, as of experimental measurement, which
++ need to be smoothed out. To seek an approximation to the data, it
++ is first necessary to specify for the approximating function a
++ mathematical form (a polynomial, for example) which contains a
++ number of unspecified coefficients: the appropriate fitting
++ routine then derives for the coefficients the values which
++ provide the best fit of that particular form. The package deals
++ mainly with curve and surface fitting (i.e., fitting with
++ functions of one and of two variables) when a polynomial or a
++ cubic spline is used as the fitting function, since these cover
++ the most common needs. However, fitting with other functions
++ and/or more variables can be undertaken by means of general
++ linear or nonlinear routines (some of which are contained in
++ other packages) depending on whether the coefficients in the
++ function occur linearly or nonlinearly. Cases where a graph
++ rather than a set of data points is given can be treated simply
++ by first reading a suitable set of points from the graph.
++ The package also contains routines for evaluating,
++ differentiating and integrating polynomial and spline curves and
++ surfaces, once the numerical values of their coefficients have
NagFittingPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage
Exports ==
  e02adf : (Integer,Integer,Integer,Matrix DoubleFloat,
            Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
  ++ e02adf(m,kplus1,nrows,x,y,w,ifail)
  ++ computes weighted least-squares polynomial approximations
  ++ to an arbitrary set of data points.
  ++ See \downlink{Manual Page}{manpageXXe02adf}.
  e02aef : (Integer,Matrix DoubleFloat,DoubleFloat,Integer) -> Result
  ++ e02aef(nplus1,a,xcap,ifail)
  ++ evaluates a polynomial from its Chebyshev-series
  ++ representation.
  ++ See \downlink{Manual Page}{manpageXXe02aef}.
  e02agf : (Integer,Integer,Integer,DoubleFloat,
            DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,
            Matrix DoubleFloat,Integer,Matrix DoubleFloat,Matrix DoubleFloat,
            Integer,Integer,Integer,Integer) -> Result
  ++ e02agf(m,kplus1,nrows,xmin,xmax,x,y,w,mf,xf,yf,lyf,ip,lwrk,liwrk,ifail)
  ++ computes constrained weighted least-squares polynomial
  ++ approximations in Chebyshev-series form to an arbitrary set of
  ++ data points. The values of the approximations and any number of
  ++ their derivatives can be specified at selected points.
  ++ See \downlink{Manual Page}{manpageXXe02agf}.
  e02ahf : (Integer,Integer,Integer,Integer,Integer) -> Result
  ++ e02ahf(np1,xmin,xmax,a,ia1,la1,ladif1,ladif,ifail)
  ++ determines the coefficients in the Chebyshev-series
  ++ representation of the derivative of a polynomial given in
  ++ Chebyshev-series form.
  ++ See \downlink{Manual Page}{manpageXXe02ahf}.
  e02ajf : (Integer,Integer,Integer,Integer,Integer) -> Result
  ++ e02ajf(np1,xmin,xmax,a,ia1,la,qatm1,iaint1,laint,ifail)
  ++ determines the coefficients in the Chebyshev-series
  ++ representation of the indefinite integral of a polynomial given
  ++ in Chebyshev-series form.
  ++ See \downlink{Manual Page}{manpageXXe02ajf}.
  e02akf : (Integer,Integer,Integer,Integer) -> Result
  ++ e02akf(np1,xmin,xmax,a,ia1,la,x,ifail)
  ++ evaluates a polynomial from its Chebyshev-series
  ++ representation, allowing an arbitrary index increment for
  ++ accessing the array of coefficients.
  ++ See \downlink{Manual Page}{manpageXXe02akf}.
  e02baf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,
Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
++ e02baf(m, ncap7, x, y, w, lamda, ifail)
++ computes a weighted least-squares approximation to an
++ arbitrary set of data points by a cubic spline
++ prescribed by the user. Cubic spline can also be
++ carried out.
++ See \downlink{Manual Page}{manpageXXe02baf}.

e02bbf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer) -> Result
++ e02bbf(ncap7, lamda, c, x, ifail)
++ evaluates a cubic spline representation.
++ See \downlink{Manual Page}{manpageXXe02bbf}.

e02bcf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, Integer) -> Result
++ e02bcf(ncap7, lamda, c, x, left, ifail)
++ evaluates a cubic spline and its first three derivatives
++ from its B-spline representation.
++ See \downlink{Manual Page}{manpageXXe02bcf}.

e02bdf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
++ e02bdf(ncap7, lamda, c, ifail)
++ computes the definite integral from its
++ B-spline representation.
++ See \downlink{Manual Page}{manpageXXe02bdf}.

e02bef : (String, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Integer, Integer, Integer, Integer, Integer, Matrix DoubleFloat, Integer, Matrix DoubleFloat, Matrix Integer) -> Result
++ e02bef(start, m, x, y, w, s, nest, lwk, n, lamda, ifail, wrk, iwrk)
++ computes a cubic spline approximation to an arbitrary set
++ of data points. The knot are located
++ automatically, but a single parameter must be specified to
++ control the trade-off between closeness of fit and smoothness of
++ fit.
++ See \downlink{Manual Page}{manpageXXe02bef}.

e02daf : (Integer, Integer, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Matrix Integer, Integer, Integer, Integer, Integer, DoubleFloat, Matrix DoubleFloat, Integer) -> Result
++ e02daf(m, px, py, x, y, f, w, mu, point, n, eps, lamda, ifail)
++ forms a minimal, weighted least-squares bicubic spline
++ surface fit with prescribed knots to a given set of data points.
++ See \downlink{Manual Page}{manpageXXe02daf}.

e02dcf : (String, Integer, Matrix DoubleFloat, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer, Integer, Integer, Integer, Integer, Matrix DoubleFloat, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix Integer, Integer) -> Result
++ e02dcf(start, mx, x, my, y, f, s, mxest, nyest, lwk, iwrk, nx, yest, lamda, mu, ifail)
++ computes a bicubic spline approximation to a set of data
values, given on a rectangular grid in the x-y plane. The knots of the spline are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

See \downlink{Manual Page}{manpageXXe02dcf}.

\begin{Verbatim}
e02ddf : (String,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer) -> Result
\end{Verbatim}

\begin{Verbatim}
e02ddf(start,m,x,y,f,w,s,nxest,nyest,lwrk,liwrk,nx,_
    lamda,ny,mu,wrk,ifail)
\end{Verbatim}

computes a bicubic spline approximation to a set of scattered data are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

See \downlink{Manual Page}{manpageXXe02ddf}.

\begin{Verbatim}
e02def : (Integer,Integer,Integer,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Integer) -> Result
\end{Verbatim}

\begin{Verbatim}
e02def(m,px,py,x,y,lamda,mu,c,ifail)
\end{Verbatim}

calculates values of a bicubic spline representation.

See \downlink{Manual Page}{manpageXXe02def}.

\begin{Verbatim}
e02dff : (Integer,Integer,Integer,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Integer,Integer,Integer) -> Result
\end{Verbatim}

\begin{Verbatim}
e02dff(mx,my,px,py,x,y,lamda,mu,c,lwrk,liwrk,ifail)
\end{Verbatim}

calculates values of a bicubic spline representation. The spline is evaluated at all points on a rectangular grid.

See \downlink{Manual Page}{manpageXXe02dff}.

\begin{Verbatim}
e02gaf : (Integer,Integer,Integer,DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
\end{Verbatim}

\begin{Verbatim}
e02gaf(m,la,nplus2,toler,a,b,ifail)
\end{Verbatim}

calculates an l solution to an over-determined system of linear equations.

See \downlink{Manual Page}{manpageXXe02gaf}.

\begin{Verbatim}
e02zaf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    Integer,Integer) -> Result
\end{Verbatim}

\begin{Verbatim}
e02zaf(px,py,lamda,mu,m,x,y,npoint,nadres,ifail)
\end{Verbatim}

sorts two-dimensional data into rectangular panels.

See \downlink{Manual Page}{manpageXXe02zaf}.

Implementation ==> add

\begin{Verbatim}
import Lisp
import DoubleFloat
import Any
\end{Verbatim}
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix Integer)
import AnyFunctions1(String)

e02adf(mArg:Integer,kplus1Arg:Integer,nrowsArg:Integer,_,
        xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_,
        wArg:Matrix DoubleFloat,_,
        ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
        "e02adf",_
        "w"::S,"a"::S,"s"::S,_,
        "work1"::S,"work2"::S]$Lisp,_,
        ["a"::S,"m"::S,"work1"::S,"work2"::S]$Lisp,_,
        ["double"::S,"x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_,
        ["w"::S,"m"::S]$Lisp,["a"::S,"nrows"::S,"kplus1"::S]$Lisp,_,
        ["m"::S,"kplus1"::S]$Lisp,["work1"::S,_,
        ["*="::S,3$Lisp,"m"::S]$Lisp]$Lisp$Lisp_,
        ["work2"::S,[*::S,2$Lisp,"kplus1"::S]$Lisp]$Lisp$Lisp,_,
        ["integer"::S,"m"::S,"kplus1"::S,"nrows"::S,_,
        "ifail"::S]$Lisp_]
)$Lisp_,

e02aef(nplus1Arg:Integer,aArg:Matrix DoubleFloat,xcapArg:DoubleFloat,_,
        ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
        "e02aef",_
        ["p"::S]$Lisp,_,
        ["double"::S,"a"::S,"nplus1"::S]$Lisp,"xcap"::S,_,
        ["p"::S]$Lisp$_,
        ["integer"::S,"nplus1"::S,"ifail"::S]$Lisp$_
)$Lisp_,

pretend List (Record(key:Symbol,entry:Any))$Result
e02agf(mArg::Integer,kplus1Arg::Integer,nrowsArg::Integer,_
  xminArg::DoubleFloat,xmaxArg::DoubleFloat,xArg::Matrix DoubleFloat,_
  yArg::Matrix DoubleFloat,wArg::Matrix DoubleFloat,mfArg::Integer,_
  xfArg::Matrix DoubleFloat,yfArg::Matrix DoubleFloat,lyfArg::Integer,_
  ipArg::Matrix Integer,lwrkArg::Integer,livrkArg::Integer,_
  ifailArg::Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02agf",_
    ["m":S,"kplus1":S,"nrows":S,"xmin":S,"xmax":S_,
      "a":S,"s":S,"npl":S,"wrk":S]$Lisp_-
    ["double":S,"x":S,"ymax":S,"x":S,"m":S]$Lisp_-
    ,["y":S,"m":S]$Lisp,["w":S,"m":S]$Lisp,["xf":S,"mf":S]$Lisp_-
    ,["s":S,"kplus1":S]$Lisp,["wrk":S,"ivrk":S]$Lisp]$Lisp_-
    [[integer":S,"m":S,"kplus1":S,"nrows":S_,
    [[[mArg::Any,kplus1Arg::Any,nrowsArg::Any,xminArg::Any,_
      xmaxArg::Any,mfArg::Any,lyfArg::Any,lwrkArg::Any,lyfArg::Any,_
      ifailArg::Any,xArg::Any,yArg::Any,xfArg::Any,_
      ifailArg::Any,ipArg::Any ]]_--
    @List Any]$Lisp)$Lisp_-
  pretend List (Record(key:Symbol,entry:Any)))$Result

e02ahf(np1Arg::Integer,xminArg::DoubleFloat,xmaxArg::DoubleFloat,_
  aArg::Matrix DoubleFloat,ialArg::Integer,laArg::Integer,_
  laddirArg::Integer,ladfArg::Integer,ifailArg::Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02ahf",_
    [["patm1":S,"adif":S]$Lisp_-
      [["double":S,"xmin":S,"xmax":S,["a":S,"la":S]$Lisp_-
        ,"patm1":S,"ladf":S]$Lisp_-
        ,["integer":S,"np1":S,"npl":S,"ladif":S,$Lisp_-
          ,"ladf":S,"ifail":S]$Lisp_-
        $Lisp_-
      $Lisp_-
    [["patm1":S,"adif":S,"ifail":S]$Lisp_-
      [[[nplArg::Any,xminArg::Any,xmaxArg::Any,ialArg::Any,laArg::Any,_
        laddirArg::Any,ladfArg::Any,ifailArg::Any,aArg::Any ]]_--
    @List Any]$Lisp)$Lisp_-
  pretend List (Record(key:Symbol,entry:Any)))$Result


```
\[
\begin{align*}
e02ajf\{(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,\
aArg:Matrix\ Double\ Float,ia1Arg:Integer,laArg:Integer,\
qatmlArg:Double\ Float,iaint1Arg:Integer,laintArg:Integer,\
ifailArg:Integer)\rightarrow\text{Result} ==
& \text{[\text{invokeNagman}(NIL$Lisp,}\
& \text{"e02ajf",}\
& \text{["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S,}\
& \text{["aint"::S]\$Lisp,}\
& \text{[["double"::S,"xmin"::S,"xmax"::S,["a"::S,"la"::S]\$Lisp,}\
& \text{"qatml"::S,["aint"::S,"laint"::S]\$Lisp]\$Lisp,}\
& \text{,["integer"::S,"np1"::S,"ia1"::S,"la"::S,"iaint1"::S,}\
& \text{"laint"::S,"ifail"::S]\$Lisp]\$Lisp,}\
& \text{["aint"::S,"ifail"::S]\$Lisp,}\
& \text{[[np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,}\
& \text{qatmlArg::Any,iaint1Arg::Any,laintArg::Any,ifailArg::Any,aArg::Any ]]}\}\
& \text{\text{\@List\ Any}\$Lisp)\$Lisp,}\
& \text{pretend\ List\ (Record(key:Symbol,entry:Any))}\$Result
\end{align*}
\]
\[
\begin{align*}
e02akf\{(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,\
aArg:Matrix\ Double\ Float,ia1Arg:Integer,laArg:Integer,\
xArg:DoubleFloat,ifailArg:Integer)\rightarrow\text{Result} ==
& \text{[\text{invokeNagman}(NIL$Lisp,}\
& \text{"e02akf",}\
& \text{["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S,}\
& \text{"x"::S,"result"::S,"ifail"::S,"a"::S]\$Lisp,}\
& \text{["result"::S]\$Lisp,}\
& \text{[["double"::S,"xmin"::S,"xmax"::S,["a"::S,"la"::S]\$Lisp,}\
& \text{"x"::S,"result"::S]\$Lisp,}\
& \text{,["integer"::S,"np1"::S,"ia1"::S,"la"::S,"ifail"::S]\$Lisp]\$Lisp,}\
& \text{["result"::S,"ifail"::S]\$Lisp,}\
& \text{[[np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,}\
& \text{xArg::Any,ifailArg::Any,aArg::Any ]]}\}\
& \text{\text{\@List\ Any}\$Lisp)\$Lisp,}\
& \text{pretend\ List\ (Record(key:Symbol,entry:Any))}\$Result
\end{align*}
\]
\[
\begin{align*}
e02baf\{(mArg:Integer,ncap7Arg:Integer,xArg:Matrix\ Double\ Float,\
yArg:Matrix\ Double\ Float,wArg:Matrix\ Double\ Float,\
lamdaArg:Matrix\ Double\ Float,ifailArg:Integer)\rightarrow\text{Result} ==
& \text{[\text{invokeNagman}(NIL$Lisp,}\
& \text{"e02baf",}\
& \text{"c"::S,"lamda"::S,}\
& \text{"work1"::S,"work2"::S]\$Lisp,}\
& \text{["c"::S,"ss"::S,"work1"::S,"work2"::S]\$Lisp,}\
& \text{[["double"::S,"x"::S,"m"::S]\$Lisp,["y"::S,"m"::S]\$Lisp,}\
& \text{[["w"::S,"m"::S]\$Lisp,["c"::S,"ncap7"::S]\$Lisp,"ss"::S,}\
& \text{\text{\@List\ Any}\$Lisp)\$Lisp,}\
& \text{pretend\ List\ (Record(key:Symbol,entry:Any))}\$Result}
\end{align*}
\]
```
e02bbf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
cArg:Matrix DoubleFloat,xArg:DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"e02bbf",_
["s":S]$Lisp,_
["double":S,["lamda":S,"ncap7":S]$Lisp_,
["c":S,"ncap7":S]$Lisp,"x":S,"s":S]$Lisp_,
["integer":S,"ncap7":S,"ifail":S]$Lisp_]
)$Lisp_,
["s":S,"ifail":S]$Lisp_,
[(ncap7Arg::Any,xArg::Any,ifailArg::Any,lamdaArg::Any,cArg::Any ])_
@List Any$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

e02bcf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
xArg:DoubleFloat,leftArg:Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"e02bcf",_
["ncap7":S,"x":S,"left":S,"ifail":S,"lamda":S,_
"c":S,"s":S]$Lisp,_
["s":S]$Lisp,_
["double":S,["lamda":S,"ncap7":S]$Lisp_,
["c":S,"ncap7":S]$Lisp,"x":S,["s":S,4$Lisp]$Lisp]$Lisp_,
["integer":S,"ncap7":S,"left":S,"ifail":S_]
]$Lisp_]
)$Lisp_,
["s":S,"ifail":S]$Lisp_,
[(ncap7Arg::Any,xArg::Any,leftArg::Any,ifailArg::Any,_
lamdaArg::Any,cArg::Any ])_
@List Any$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

e02bdf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
cArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"e02bdf",_
CHAPTER 15. CHAPTER N

["defint"::S]$Lisp, 
["defint"::S]$Lisp, 
[([ncap7Arg::Any,ifailArg::Any,lamdaArg::Any,cArg::Any ])_ 
@List Any]$Lisp]$Lisp)_ 
pretend List (Record(key:Symbol,entry:Any))$Result

e02bef(startArg::String,mArg:Integer,xArg:Matrix DoubleFloat, _
yArg:Matrix DoubleFloat,wArg:Matrix DoubleFloat,sArg:DoubleFloat,_
nestArg:Integer,lwrkArg:Integer,nArg:Integer,_
lamdaArg:Matrix DoubleFloat,ifailArg:Integer,_
wrkArg:Matrix DoubleFloat, _
iwrkArg:Matrix Integer): Result == 
[(invokeNagman(NIL$Lisp,_
"e02bef",_
["c"::S,"fp"::S]$Lisp, _
,["character"::S,"start"::S]$Lisp_ ]$Lisp_ 
,$Lisp_ _
[(startArg::Any,mArg::Any,sArg::Any,nestArg::Any,lwrkArg::Any,_
nArg::Any,ifailArg::Any,xArg::Any,yArg::Any,wArg::Any,_
lamdaArg::Any,xrArg::Any,lwrkArg::Any,ifailArg::Any )]
@List Any]$Lisp]$Lisp)_ 
pretend List (Record(key:Symbol,entry:Any))$Result

e02daf(mArg:Integer,pxArg:Integer,pyArg:Integer, _
xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, _
wArg:Matrix DoubleFloat,muArg:Matrix DoubleFloat, _
pointArg:Matrix Integer, _
npointArg:Integer,ncArg:Integer,nwsArg:Integer, _
epsArg:DoubleFloat,lamdaArg:Matrix DoubleFloat, _
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp,_
"e02daf",_
"x"::S,"y"::S,"f"::S,"w"::S,"mu"::S_ 
,point"::S,"d1"::S,"c"::S,"lamda"::S,"ws"::S_ 
]@List Any)$Lisp, 
["d1"::S,"c"::S,"sigma"::S,"rank"::S,"ws"::S]$Lisp, 
["double"::S,"x"::S,"m"::S]$Lisp, ["y"::S,"m"::S]$Lisp_ 
,["f"::S,"m"::S]$Lisp,["w"::S,"m"::S]$Lisp, 
["mu"::S,"py"::S]$Lisp, "eps"::S,"[d1"::S,"nc"::S]$Lisp, 
["c"::S,"nc"::S]$Lisp 
]$Lisp, 
[[mArg::Any,pxArg::Any,pyArg::Any,npointArg::Any,ncArg::Any, 
 nwsArg::Any,epsArg::Any,ifailArg::Any,xArg::Any,yArg::Any, 
 fArg::Any,yArg::Any,pointArg::Any,lamdaArg::Any, 
@ifailArg::Integer): Result == 
 [(invokeNagman(NIL$Lisp, 
 "e02dcf","start"::S,"mx"::S,"my"::S,"s"::S,"nxest"::S_ 
 ,"mu"::S,"lwkrk"::S,"lwkrk"::S]$Lisp, 
 ["c"::S,"fp"::S]$Lisp, 
 ,["f"::S,"m"::S,"mx"::S,"my"::S]$Lisp]$Lisp, ["f"::S, 
 ,["iwrk"::S,"ifail"::S]$Lisp, 
 ,["integer"::S,"m"::S,"my"::S,"nxest"::S,"nyest"::S_ 
 ,"ifail"::S]$Lisp_ 
 ,["character"::S,"start"::S]$Lisp_ 
 ]]$Lisp_ 
 "lwkrk"::S,"ifail"::S]$Lisp_ 
 [[[startArg::Any,mxArg::Any,myArg::Any,sArg::Any,nxestArg::Any, 
 nyestArg::Any,lwkrkArg::Any,iwrkArg::Any,nyArg::Any, 
 ifailArg::Any,xArg::Any,yArg::Any,fArg::Any,lamdaArg::Any, 

CHAPTER 15. CHAPTER N

\begin{verbatim}
muArg::Any,wrkArg::Any,iwrkArg::Any ]\_ 
@List Any\[$Lisp]$Lisp\_ 
pretend List (Record(key:Symbol,entry:Any))\]$Result 
e02ddf(startArg:String,mArg:Integer,xArg:Matrix DoubleFloat, 
  yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, 
  wArg:Matrix DoubleFloat, 
  sArg:DoubleFloat,nxestArg:Integer, 
  nyestArg:Integer, 
  lwrkArg:Integer,liwrkArg:Integer, 
  lamdaArg:Matrix DoubleFloat,nyArg:Integer,muArg:Matrix DoubleFloat, 
  wrkArg:Matrix DoubleFloat,ifailArg:Integer): Result == 
  [(invokeNagman(NIL$Lisp, 
    "e02ddf", 
  [["double"::S,"x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_, 
    ["fp"::S,"m"::S]$Lisp,["w"::S,"m"::S]$Lisp,"s"::S_, 
    ["c"::S,["*"::S,"nxest"::S,4$Lisp]$Lisp,_, 
    ["-"::S,"nyest"::S,4$Lisp]$Lisp]$Lisp$_ 
    ["wrk"::S,"iwrk"::S]$Lisp$_ 
  ]\]$Lisp,_, 
  "nx"::S,"ny"::S,"ifail"::S]$Lisp_,_, 
[["character"::S,"start"::S]$Lisp$_ 
  ]\]$Lisp,_, 
  "mu"::S,"wrk"::S,"iwrk"::S]$Lisp,_, 
  [["startArg::Any,mArg::Any,sArg::Any,nxestArg::Any,nyestArg::Any_, 
    lwrkArg::Any,iwrkArg::Any,nyArg::Any,nyArg::Any,muArg::Any_, 
    wrkArg::Any ]\_ 
  @List Any\[$Lisp]$Lisp\_ 
pretend List (Record(key:Symbol,entry:Any))\]$Result 
e02def(mArg:Integer,pxArg:Integer,pyArg:Integer, 
  xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat, 
  lamdaArg:Matrix DoubleFloat, 
  muArg:Matrix DoubleFloat,cArg:Matrix DoubleFloat, 
  ifailArg:Integer): Result == 
  [(invokeNagman(NIL$Lisp, 
    "e02def", 
    ["ff"::S,"wrk"::S,"iwrk"::S]$Lisp,_, 
  ]\]$Lisp)\]$Lisp 
\end{verbatim}


```
[["double"::S,"x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_  
,["lamda"::S,"px"::S]$Lisp,["mu"::S,"py"::S]$Lisp,_  
,"c"::S,["*"::S,"-"::S,"px"::S,4$Lisp]$Lisp,_  
,["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_  
,["ff"::S,"m"::S]$Lisp,["wrk"::S,y_  
,["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_  
,["wrk"::S,["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp]_  
]$Lisp_  
["ff"::S,"fail"::S]$Lisp_  
[([mArg::Any,pxArg::Any,pyArg::Any,ifailArg::Any,xArg::Any,_  
yArg::Any,lamdaArg::Any,muArg::Any,cArg::Any ])_  
@List Any]$Lisp]$_Lisp_  
pretend List (Record(key:Symbol,entry:Any))$Result

```

```lisp
(e02dff (mxArg:Integer,myArg:Integer,pxArg:Integer,_  
pyArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
 lamdaArg:Matrix DoubleFloat,muArg:Matrix DoubleFloat,_  
 cArg:Matrix DoubleFloat,_  
   [(invokeNagman(NIL$Lisp,_  
 "e02dff",_,
["ff"::S,"wrk"::S,"iwrk"::S]$Lisp_,  
,["lamda"::S,"px"::S]$Lisp,["mu"::S,"py"::S]$Lisp,_  
,["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_  
,["wrk"::S,"lwrk"::S]$Lisp]$Lisp_,  
]$Lisp_  
["ff"::S,"ifail"::S]$Lisp_  
[([mxArg::Any,myArg::Any,pxArg::Any,pyArg::Any,lwrkArg::Any,_  
 iwrkArg::Any,ifailArg::Any,xArg::Any,yArg::Any,lamdaArg::Any,_  
 muArg::Any,cArg::Any ])_  
@List Any]$Lisp]$_Lisp_  
pretend List (Record(key:Symbol,entry:Any))$Result

```

```lisp
(e02gaf (mArg:Integer,laArg:Integer,nplus2Arg:Integer,_  
tolerArg:DoubleFloat,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_  
ifailArg:Integer): Result ==
   [(invokeNagman(NIL$Lisp,_  
 "e02gaf",_,
```

```
package NAGF04 NagLinearEquationSolvingPackage

— NagLinearEquationSolvingPackage.input —

)set break resume
)sys rm -f NagLinearEquationSolvingPackage.output
)spool NagLinearEquationSolvingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagLinearEquationSolvingPackage
--E 1

)spool
)lisp (bye)

— NagLinearEquationSolvingPackage.help —

This package uses the NAG Library to solve the matrix equation
\[ AX = B, \]
where \( B \) may be a single vector or a matrix of multiple right-hand sides.

The matrix \( A \) may be real, complex, symmetric, Hermitian positive-definite,
or sparse. It may also be rectangular, in which case a least-squares solution is obtained.

F04 -- Simultaneous Linear Equations
Introduction -- F04
Chapter F04
Simultaneous Linear Equations

1. Scope of the Chapter

This chapter, together with two routines in Chapter F07, is concerned with the solution of the matrix equation \( AX = B \), where \( B \) may be a single vector or a matrix of multiple right-hand sides. The matrix \( A \) may be real, complex, symmetric, Hermitian positive-definite, or sparse. It may also be rectangular, in which case a least-squares solution is obtained.

2. Background to the Problems

A set of linear equations may be written in the form
where the known matrix $A$, with real or complex coefficients, is of size $m$ by $n$, ($m$ rows and $n$ columns), the known right-hand vector $b$ has $m$ components ($m$ rows and one column), and the required solution vector $x$ has $n$ components ($n$ rows and one column). There may sometimes be $p$ vectors $b_i$, $i=1,2,...,p$ on the right-hand side and the equations may then be written as

$$AX = B$$

the required matrix $X$ having as its $p$ columns the solutions of $Ax = b_i$, $i=1,2,...,p$. Some routines deal with the latter case, but for clarity only the case $p=1$ is discussed here.

The most common problem, the determination of the unique solution of $Ax=b$, occurs when $m=n$ and $A$ is non-singular, that is $\text{rank}(A)=n$ problem, discussed in Section 2.2 below, is the determination of the least-squares solution of $Ax\approx=b$, i.e., the determination of a vector $x$ which minimizes the Euclidean length (two norm) of the residual vector $r=b-Ax$. The usual case has $m>n$ and $\text{rank}(A)=n$, in which case $x$ is unique.

2.1. Unique Solution of $Ax=b$

Most of the routines in this chapter, as well as two routines in Chapter F07, solve this particular problem. The solution is obtained by performing either an LU factorization, or a Cholesky factorization, as discussed in Section 2 of the F01 Chapter Introduction.

Two of the routines in this chapter use a process called iterative refinement to improve the initial solution in order to obtain a solution that is correct to working accuracy. It should be emphasised that if $A$ and $b$ are not known exactly then not all the figures in this solution may be meaningful. To be more precise, if $x$ is the exact solution of the equations $Ax=b$

and $x$ is the solution of the perturbed equations

$$(A+E)x=b+e,$$
then, provided that \((\kappa(A) \leq 1)
\)

\[
\frac{\|x - x^\star\|}{\|x\|} \leq \frac{(\kappa(A) \|E\| \|e\|)}{1 - (\kappa(A) \frac{\|A\|}{\|A\|^{-1}})}.
\]

where \((\kappa(A) = \|A\|\|A^{-1}\|)\) is the condition number of \(A\) with respect to inversion. Thus, if \(A\) is ill-conditioned (\(\kappa(A)\) is large), \(x\) may differ significantly from \(x^\star\). Often

\[
\frac{(\kappa(A) \frac{\|A\|}{\|A\|^{-1}})\|E\| \|e\|}{\|x\|} \ll 1
\]

in which case the above bound effectively reduces to

\[
\frac{\|x - x^\star\|}{\|x\|} \leq \frac{(\kappa(A) \|E\| \|e\|)}{\|A\| \|b\|}.
\]

2.2. The Least-squares Solution of \(Ax^\star = b\)

The least-squares problem is to find a vector \(x\) to minimize

\[
T_r r, \text{ where } r = b - Ax.
\]

When \(m > n\) and \(\text{rank}(A) = n\) then the solution vector \(x\) is unique. For the cases where \(x\) is not unique the routines in this chapter obtain the minimal length solution, that is the vector \(x\) for which \(x^T x\) is a minimum.

2.3. Calculating the Inverse of a Matrix

The routines in this chapter can also be used to calculate the inverse of a square matrix \(A\) by solving the equation
where $I$ is the identity matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Purpose Routines

Many of the routines in this chapter perform the complete solution of the required equations, but some of the routines, as well as the routines in Chapter F07, assume that a prior factorization has been performed, using the appropriate factorization routine from Chapter F01 or Chapter F07. These, so-called, general purpose routines can be useful when explicit information on the factorization is required, as well as the solution of the equations, or when the solution is required for multiple right-hand sides, or for a sequence of right-hand sides.

Note that some of the routines that perform a complete solution also allow multiple right-hand sides.

3.2. Iterative Refinement

The routines that perform iterative refinement are more costly than those that do not perform iterative refinement, both in terms of time and storage, and should only be used if the problem really warrants the additional accuracy provided by these routines. The storage requirements are approximately doubled, while the additional time is not usually prohibitive since the initial factorization is used at each iteration.

3.3. Sparse Matrix Routines

The routines for sparse matrices should usually be used only when the number of non-zero elements is very small, less than 10% of the total number of elements of $A$. Additionally, when the matrix is symmetric positive-definite the sparse routines should generally be used only when $A$ does not have a (variable) band structure.

There are four routines for solving sparse linear equations, two for solving general real systems (F04AXF and F04QAF), one for solving symmetric positive-definite systems (F04MAF) and one for solving symmetric systems that may, or may not, be positive-definite (F04MBF). F04AXF and F04MAF utilise factorizations of the matrix $A$ obtained by routines in Chapter F01, while the other two routines use iterative techniques and require a user-supplied $T$ function to compute matrix-vector products $Ac$ and $A^Tc$ for any given vector $c$. The routines requiring factorizations will
usually be faster and the factorization can be utilised to solve for several right-hand sides, but the original matrix has to be explicitly supplied and is overwritten by the factorization, and the storage requirements will usually be substantially more than those of the iterative routines.

Routines F04MBF and F04QAF both allow the user to supply a pre-conditioner.

F04MBF can be used to solve systems of the form \((A-\lambda I)x=b\), which can be useful in applications such as Rayleigh quotient iteration.

F04QAF also solves sparse least-squares problems and allows the solution of damped (regularized) least-squares problems.

3.4. Decision Trees

If at any stage the answer to a question is 'Don't know' this should be read as 'No'.

For those routines that need to be preceded by a factorization routine, the appropriate routine name is given in brackets after the name of the routine for solving the equations. Note also that you may be directed to a routine in Chapter F07.

3.4.1. Routines for unique solution of \(Ax=b\)

Please see figure in printed Reference Manual

3.4.2. Routines for Least-squares problems

Please see figure in printed Reference Manual

F04 -- Simultaneous Linear Equations

Chapter F04

Eigenvalues and Eigenvectors

F04ADF Approximate solution of complex simultaneous linear equations with multiple right-hand sides
F04ARF Approximate solution of real simultaneous linear equations, one right-hand side

F04ASF Accurate solution of real symmetric positive-definite simultaneous linear equations, one right-hand side

F04ATF Accurate solution of real simultaneous linear equations, one right-hand side

F04AXF Approximate solution of real sparse simultaneous linear equations (coefficient matrix already factorized by F01BRF or F01BSF)

F04FAF Approximate solution of real symmetric positive-definite tridiagonal simultaneous linear equations, one right-hand side

F04JGF Least-squares (if rank = n) or minimal least-squares (if rank < n) solution of m real equations in n unknowns, rank <= n, m >= n

F04MAF Real sparse symmetric positive-definite simultaneous linear equations (coefficient matrix already factorized)

F04MBF Real sparse symmetric simultaneous linear equations

F04MCF Approximate solution of real symmetric positive-definite variable-bandwidth simultaneous linear equations (coefficient matrix already factorized)

F04QAF Sparse linear least-squares problem, m real equations in n unknowns

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F04 -- Simultaneous Linear Equations F04ADF
F04ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ADF calculates the approximate solution of a set of complex linear equations with multiple right-hand sides, using an LU factorization with partial pivoting.

2. Specification
SUBROUTINE F04ADF (A, IA, B, IB, N, M, C, IC, WKSPCE, IFAIL)
  INTEGER IA, IB, N, M, IC, IFAIL
  DOUBLE PRECISION WKSPCE(*)
  COMPLEX(KIND(1.0D0)) A(IA,*), B(IB,*), C(IC,*)

3. Description

Given a set of complex linear equations AX=B, the routine first computes an LU factorization of A with partial pivoting, PA=LU, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. The columns x of the solution X are found by forward and backward substitution in Ly=Pb and Ux=y, where b is a column of the right-hand side matrix B.

4. References


5. Parameters

1: A(IA,*), B(IB,*), N -- COMPLEX(KIND(1.0D0)) array
   Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the n by n matrix A. On exit: A is overwritten by the lower triangular matrix L and the off-diagonal elements of the upper triangular matrix U. The unit diagonal elements of U are not stored.

2: IA -- INTEGER
   Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F04ADF is called.
   Constraint: IA >= max(1,N).

3: IB -- INTEGER
   Input
   On entry: the first dimension of the array B as declared in the (sub)program from which F04ADF is called.
   Constraint: IB >= max(1,N).

4: N -- INTEGER
   Input
On entry: \( n \), the order of the matrix \( A \). Constraint: \( N \geq 0 \).

6: \( M \) -- INTEGER Input
On entry: \( m \), the number of right-hand sides. Constraint: \( M \geq 0 \).

7: \( C(IC,*\) -- COMPLEX(KIND(1.0D)) array Output
Note: the second dimension of the array \( C \) must be at least \( \max(1, M) \).
On exit: the \( n \) by \( m \) solution matrix \( X \). See also Section 8.

8: \( IC \) -- INTEGER Input
On entry:
the first dimension of the array \( C \) as declared in the
(sub)program from which F04ADF is called.
Constraint: \( IC \geq \max(1, N) \).

9: \( WKSPCE(*) \) -- DOUBLE PRECISION array Workspace
Note: the dimension of the array \( WKSPCE \) must be at least \( \max(1, N) \).

10: \( IFAIL \) -- INTEGER Input/Output
On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: \( IFAIL = 0 \) unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \( IFAIL = 0 \) or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

\( IFAIL = 1 \)
The matrix \( A \) is singular, possibly due to rounding errors.

\( IFAIL = 2 \)
On entry \( N < 0 \),
or \( M < 0 \),
or \( IA < \max(1, N) \),
or \( IB < \max(1, N) \),
or \( IC < \max(1, N) \).
7. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see Wilkinson and Reinsch [1] page 106.

8. Further Comments

The time taken by the routine is approximately proportional to $n$

Unless otherwise stated in the Users’ Note for your implementation, the routine may be called with the same actual array supplied for parameters B and C, in which case the solution vectors will overwrite the right-hand sides. However this is not standard Fortran 77, and may not work on all systems.

9. Example

To solve the set of linear equations $AX=B$ where

$$
A = \begin{pmatrix}
1 & 1+2i & 2+10i \\
1+i & 3i & -5+14i \\
1+i & 5i & -8+20i
\end{pmatrix}
$$

and

$$
B = \begin{pmatrix}
1 \\
0 \\
0
\end{pmatrix}
$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
2. Specification

```fortran
SUBROUTINE F04ARF (A, IA, B, N, C, WKSPCE, IFAIL)
INTEGER IA, N, IFAIL
DOUBLE PRECISION A(IA,*), B(*), C(*), WKSPCE(*)
```

3. Description

Given a set of linear equations, $Ax=b$, the routine first computes an LU factorization of $A$ with partial pivoting, $PA=LU$, where $P$ is a permutation matrix, $L$ is lower triangular and $U$ is unit upper triangular. The approximate solution $x$ is found by forward and backward substitution in $Ly=Pb$ and $Ux=y$, where $b$ is the right-hand side.

4. References


5. Parameters

1: $A(IA,*)$ -- DOUBLE PRECISION array
   Input/Output
   Note: the second dimension of the array $A$ must be at least $\max(1,N)$.
   On entry: the $n$ by $n$ matrix $A$. On exit: $A$ is overwritten by the lower triangular matrix $L$ and the off-diagonal elements of the upper triangular matrix $U$. The unit diagonal elements of $U$ are not stored.

2: $IA$ -- INTEGER
   Input
   On entry: the first dimension of the array $A$ as declared in the (sub)program from which F04ARF is called.
   Constraint: $IA \geq \max(1,N)$.

3: $B(*)$ -- DOUBLE PRECISION array
   Input
   Note: the dimension of the array $B$ must be at least $\max(1,N)$.
   On entry: the right-hand side vector $b$.

4: $N$ -- INTEGER
   Input
   On entry: $n$, the order of the matrix $A$. Constraint: $N \geq 0$.

5: $C(*)$ -- DOUBLE PRECISION array
   Output
   Note: the dimension of the array $C$ must be at least $\max(1,N)$.
   On exit: the solution vector $x$. 
6: WKSPCE(*) -- DOUBLE PRECISION array
    Workspace
    Note: the dimension of the array WKSPCE must be at least
    max(1,N).

7: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
    The matrix A is singular, possibly due to rounding errors.

IFAIL= 2
    On entry N < 0,
    or IA < max(1,N).

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters B and C, in which case the solution
vector will overwrite the right-hand side. However this is not
standard Fortran 77, and may not work on all systems.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{pmatrix}
33 & 16 & 72
\end{pmatrix}
\]
\[ A = \begin{pmatrix} -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix} \]

and

\[ b = \begin{pmatrix} -359 \\ 281 \end{pmatrix} \begin{pmatrix} 85 \end{pmatrix} \]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

\begin{verbatim}
F04 -- Simultaneous Linear Equations
F04ASF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ASF calculates the accurate solution of a set of real symmetric positive-definite linear equations with a single right-hand side, \( Ax = b \), using a Cholesky factorization and iterative refinement.

2. Specification

\begin{verbatim}
SUBROUTINE F04ASF (A, IA, B, N, C, WK1, WK2, IFAIL)
INTEGER IA, N, IFAIL
DOUBLE PRECISION A(IA,*), B(*), C(*), WK1(*), WK2(*)
\end{verbatim}

3. Description

Given a set of real linear equations \( Ax = b \), where \( A \) is a symmetric positive-definite matrix, the routine first computes a Cholesky factorization of \( A \) as \( A = LL^T \) where \( L \) is lower triangular. An approximation to \( x \) is found by forward and backward substitution. The residual vector \( r = b - Ax \) is then calculated using additional \( T \) precision and a correction \( d \) to \( x \) is found by solving \( LL^T d = r \). \( x \) is then replaced by \( x + d \), and this iterative refinement of the solution is repeated until machine accuracy is obtained.
\end{verbatim}
4. References


5. Parameters

1: A(IA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the upper triangle of the n by n positive-definite symmetric matrix A. The elements of the array below the diagonal need not be set. On exit: the elements of the array below the diagonal are overwritten; the upper triangle of A is unchanged.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F04ASF is called.
   Constraint: IA >= max(1,N).

3: B(*) -- DOUBLE PRECISION array Input
   Note: the dimension of the array B must be at least max(1,N).
   On entry: the right-hand side vector b.

4: N -- INTEGER Input
   On entry: n, the order of the matrix A. Constraint: N >= 0.

5: C(*) -- DOUBLE PRECISION array Output
   Note: the dimension of the array C must be at least max(1,N).
   On exit: the solution vector x.

6: WK1(*) -- DOUBLE PRECISION array Workspace
   Note: the dimension of the array WK1 must be at least max(1,N).

7: WK2(*) -- DOUBLE PRECISION array Workspace
   Note: the dimension of the array WK2 must be at least max(1,N).

8: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1  
The matrix A is not positive-definite, possibly due to rounding errors.

IFAIL= 2  
Iterative refinement fails to improve the solution, i.e., the matrix A is too ill-conditioned.

IFAIL= 3  
On entry N < 0,  

or IA < max(1,N).

7. Accuracy

The computed solutions should be correct to full machine accuracy. For a detailed error analysis see Wilkinson and Reinsch [1] page 39.

8. Further Comments

The time taken by the routine is approximately proportional to n

The routine must not be called with the same name for parameters B and C.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{bmatrix}
5 & 7 & 6 & 5 \\
7 & 10 & 8 & 7 \\
6 & 8 & 10 & 9 \\
5 & 7 & 9 & 10 \\
\end{bmatrix}
\]

and

\[
\begin{bmatrix}
23 \\
32 \\
\end{bmatrix}
\]

b=\(33\).
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F04 -- Simultaneous Linear Equations

F04ATF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ATF calculates the accurate solution of a set of real linear equations with a single right-hand side, using an LU factorization with partial pivoting, and iterative refinement.

2. Specification

```fortran
SUBROUTINE F04ATF (A, IA, B, N, C, AA, IAA, WKS1, WKS2, IFAIL)
INTEGER IA, N, IAA, IFAIL
DOUBLE PRECISION A(IA,*), B(*), C(*), AA(IAA,*), WKS1(*), WKS2(*)
```

3. Description

Given a set of real linear equations, \( Ax=b \), the routine first computes an LU factorization of \( A \) with partial pivoting, \( PA=LU \), where \( P \) is a permutation matrix, \( L \) is lower triangular and \( U \) is unit upper triangular. An approximation to \( x \) is found by forward and backward substitution in \( Ly=Pb \) and \( Ux=y \). The residual vector \( r=b-Ax \) is then calculated using additional precision, and a correction \( d \) to \( x \) is found by solving \( LUd=r \). \( x \) is replaced by \( x+d \), and this iterative refinement of the solution is repeated until full machine accuracy is obtained.

4. References


5. Parameters

1: A(IA,*) -- DOUBLE PRECISION array Input
Note: the second dimension of the array A must be at least \( \max(1,N) \).
On entry: the \( n \) by \( n \) matrix \( A \).

2: IA -- INTEGER
Input
On entry:
the first dimension of the array \( A \) as declared in the
(sub)program from which F04ATF is called.
Constraint: \( IA \geq \max(1,N) \).

3: B(*) -- DOUBLE PRECISION array
Input
Note: the dimension of the array \( B \) must be at least
\( \max(1,N) \).
On entry: the right-hand side vector \( b \).

4: N -- INTEGER
Input
On entry: \( n \), the order of the matrix \( A \). Constraint: \( N \geq 0 \).

5: C(*) -- DOUBLE PRECISION array
Output
Note: the dimension of the array \( C \) must be at least
\( \max(1,N) \).
On exit: the solution vector \( x \).

6: AA(IAA,*) -- DOUBLE PRECISION array
Output
Note: the second dimension of the array \( AA \) must be at least
\( \max(1,N) \).
On exit: the triangular factors \( L \) and \( U \), except that the
unit diagonal elements of \( U \) are not stored.

7: IAA -- INTEGER
Input
On entry:
the first dimension of the array \( AA \) as declared in the
(sub)program from which F04ATF is called.
Constraint: \( IAA \geq \max(1,N) \).

8: WKS1(*) -- DOUBLE PRECISION array
Workspace
Note: the dimension of the array \( WKS1 \) must be at least
\( \max(1,N) \).

9: WKS2(*) -- DOUBLE PRECISION array
Workspace
Note: the dimension of the array \( WKS2 \) must be at least
\( \max(1,N) \).

10: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   The matrix A is singular, possibly due to rounding errors.

IFAIL = 2
   Iterative refinement fails to improve the solution, i.e., the matrix A is too ill-conditioned.

IFAIL = 3
   On entry N < 0,
      or IA < max(1,N),
      or IAA < max(1,N).

7. Accuracy

The computed solutions should be correct to full machine accuracy. For a detailed error analysis see Wilkinson and Reinsch [1] page 107.

8. Further Comments

The time taken by the routine is approximately proportional to n^3.

The routine must not be called with the same name for parameters B and C.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{pmatrix}
33 & 16 & 72 \\
-24 & -10 & -57 \\
-8 & -4 & -17
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
-359 \\
281
\end{pmatrix}
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**F04AXF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

   F04AXF calculates the approximate solution of a set of real sparse linear equations with a single right-hand side, $Ax=b$ or $^T A x=b$, where $A$ has been factorized by F01BRF or F01BSF.

2. **Specification**

   ```
   SUBROUTINE F04AXF (N, A, LICN, ICN, IKEEP, RHS, W, MTYPE,  
   1                  IDISP, RESID)
   INTEGER          N, LICN, ICN(LICN), IKEEP(5*N), MTYPE,  
   1                  IDISP(2)
   DOUBLE PRECISION  A(LICN), RHS(N), W(N), RESID
   ```

3. **Description**

   To solve a system of real linear equations $Ax=b$ or $^T A x=b$, where $A$ is a general sparse matrix, $A$ must first be factorized by F01BRF or F01BSF. F04AXF then computes $x$ by block forward or backward substitution using simple forward and backward substitution within each diagonal block.

   The method is fully described in Duff [1].

4. **References**


5. **Parameters**
1: N -- INTEGER Input
   On entry: n, the order of the matrix A.

2: A(LICN) -- DOUBLE PRECISION array Input
   On entry: the non-zero elements in the factorization of the
   matrix A, as returned by F01BRF or F01BSF.

3: LICN -- INTEGER Input
   On entry:
   the dimension of the arrays A and ICN as declared in the
   (sub)program from which F04AXF is called.

4: ICN(LICN) -- INTEGER array Input
   On entry: the column indices of the non-zero elements of
   the factorization, as returned by F01BRF or F01BSF.

5: IKEEP(5*N) -- INTEGER array Input
   On entry: the indexing information about the factorization,
   as returned by F01BRF or F01BSF.

6: RHS(N) -- DOUBLE PRECISION array Input/Output
   On entry: the right-hand side vector b. On exit: RHS is
   overwritten by the solution vector x.

7: W(N) -- DOUBLE PRECISION array Workspace

8: MTYPE -- INTEGER Input
   On entry: MTYPE specifies the task to be performed:
   
   if MTYPE = 1, solve Ax=b,

   \[ T \]
   if MTYPE \neq 1, solve A x=b.

9: IDISP(2) -- INTEGER array Input
   On entry: the values returned in IDISP by F01BRF.

10: RESID -- DOUBLE PRECISION Output
    On exit: the value of the maximum residual,
    
    \[ \max(|b - > a x |), \text{over all the unsatisfied equations, in} \]
    \[ i \text{ -- i }, j \]
    case F01BRF or F01BSF has been used to factorize a singular
    or rectangular matrix.

6. Error Indicators and Warnings

None.

7. Accuracy
The accuracy of the computed solution depends on the conditioning of the original matrix. Since F04AXF is always used with either F01BRF or F01BSF, the user is recommended to set GROW = TRUE on entry to these routines and to examine the value of W(1) on exit (see the routine documents for F01BRF and F01BSF). For a detailed error analysis see Duff [1] page 17.

If storage for the original matrix is available then the error can be estimated by calculating the residual

\[ r = b - Ax \]  (or \( b - Ax \))

and calling F04AXF again to find a correction (delta) for \( x \) by solving

\[ A(delta) = r \]  (or \( A(delta) = r \)).

8. Further Comments

If the factorized form contains (tau) non-zeros (IDISP(2) = (tau)) then the time taken is very approximately 2(tau) times longer than the inner loop of full matrix code. Some advantage is taken of zeros in the right-hand side when solving \( A x = b \) (MTRY \( \neq 1 \)).

9. Example

To solve the set of linear equations \( Ax = b \) where

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 & 0 \\
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
-2 & 0 & 0 & 1 & 1 & 0 \\
-1 & 0 & 0 & -1 & 2 & -3 \\
-1 & -1 & 0 & 0 & 0 & 6 \\
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
15 \\
12 \\
18 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
3 \\
-6 \\
0 \\
\end{pmatrix}
\]

The non-zero elements of \( A \) and indexing information are read in
by the program, as described in the document for F01BRF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose

F04FAF calculates the approximate solution of a set of real symmetric positive-definite tridiagonal linear equations.

2. Specification

```
SUBROUTINE F04FAF (JOB, N, D, E, B, IFAIL)
INTEGER JOB, N, IFAIL
DOUBLE PRECISION D(N), E(N), B(N)
```

3. Description

F04FAF is based upon the Linpack routine DPTSL (see Dongarra et al [1]) and solves the equations

\[ T x = b, \]

where \( T \) is a real \( n \) by \( n \) symmetric positive-definite tridiagonal matrix, using a modified symmetric Gaussian elimination algorithm \( T \) to factorize \( T = K M M \), where \( K \) is diagonal and \( M \) is a matrix of multipliers as described in Section 8.

When the input parameter JOB is supplied as 1, then the routine assumes that a previous call to F04FAF has already factorized \( T \); otherwise JOB must be supplied as 0.

4. References

LINPACK Users’ Guide. SIAM, Philadelphia.

5. Parameters
1: JOB -- INTEGER
On entry: specifies the job to be performed by F04FAF as follows:
JOB = 0
The matrix T is factorized and the equations Tx=b are solved for x.
JOB = 1
The matrix T is assumed to have already been factorized by a previous call to F04FAF with JOB = 0;
the equations Tx=b are solved for x.

2: N -- INTEGER
On entry: n, the order of the matrix T. Constraint: N >= 1.

3: D(N) -- DOUBLE PRECISION array
On entry: if JOB = 0, D must contain the diagonal elements of T. If JOB = 1, D must contain the diagonal matrix K, as returned by a previous call of F04FAF with JOB = 0. On exit: if JOB = 0, D is overwritten by the diagonal matrix K of the factorization. If JOB = 1, D is unchanged.

4: E(N) -- DOUBLE PRECISION array
On entry: if JOB = 0, E must contain the super-diagonal elements of T, stored in E(2) to E(n). If JOB = 1, E must contain the off-diagonal elements of the matrix M, as returned by a previous call of F04FAF with JOB = 0. E(1) is not used. On exit: if JOB = 0, E(2) to E(n) are overwritten by the off-diagonal elements of the matrix M of the factorization. If JOB = 1, E is unchanged.

5: B(N) -- DOUBLE PRECISION array
On entry: the right-hand side vector b. On exit: B is overwritten by the solution vector x.

6: IFAIL -- INTEGER
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
On entry N < 1,
or \( \text{JOB} /= 0 \) or 1.

**IFAIL= 2**

The matrix \( T \) is either not positive-definite or is nearly singular. This failure can only occur when \( \text{JOB} = 0 \) and inspection of the elements of \( D \) will give an indication of why failure has occurred. If an element of \( D \) is close to zero, then \( T \) is probably nearly singular; if an element of \( D \) is negative but not close to zero, then \( T \) is not positive-definite.

**IFAILOverflow**

If overflow occurs during the execution of this routine, then either \( T \) is very nearly singular or an element of the right-hand side vector \( b \) is very large. In this latter case the equations should be scaled so that no element of \( b \) is very large. Note that to preserve symmetry it is necessary to scale by a transformation of the form \((PTP)b=Px\), where \( P \) is a diagonal matrix.

**IFAILUnderflow**

Any underflows that occur during the execution of this routine are harmless.

7. **Accuracy**

The computed factorization (see Section 8) will satisfy the equation

\[
T = T + E
\]

where \( ||E|| \leq 2(\varepsilon)||T|| \), \( p=1,F,\infty \),

\( (\varepsilon) \) being the machine precision. The computed solution of the equations \( Tx=b \), say \( x \), will satisfy an equation of the form

\[
(T+F)x=b,
\]

where \( F \) can be expected to satisfy a bound of the form

\[
||F|| \leq (\alpha)(\varepsilon)||T||,
\]
(alpha) being a modest constant. This implies that the relative error in $x$ satisfies

$$\frac{||x-x'||}{||x||} \leq c(T)(\alpha)(\epsilon),$$

where $c(T)$ is the condition number of $T$ with respect to inversion. Thus if $T$ is nearly singular, $x$ can be expected to have a large relative error.

8. Further Comments

The time taken by the routine is approximately proportional to $n$.

The routine eliminates the off-diagonal elements of $T$ by simultaneously performing symmetric Gaussian elimination from the top and the bottom of $T$. The result is that $T$ is factorized as

$$T = MKM,$$

where $K$ is a diagonal matrix and $M$ is a matrix of the form

$$M = (\begin{array}{cccccc}
1 & 0 & 0 & \ldots & 0 & 0 \\
m & 1 & 0 & \ldots & 0 & 0 \\
2 & & & & & \\
0 & m & 1 & \ldots & 0 & 0 \\
3 & & & & & \\
\vdots & & & & & \\
0 & 0 & 0 & \ldots & 1 & 0 \\
0 & 0 & 0 & \ldots & m & 1 \\
\end{array})$$

where $j$ is the integer part of $n/2$. (For example when $n=5, j=2$.) The
diagonal elements of $K$ are returned in $D$ with $k$ in the $i$th element of $D$ and $m$ is returned in the $i$th element of $E$.

The routine fails with IFAIL = 2 if any diagonal element of $K$ is non-positive. It should be noted that $T$ may be nearly singular even if all the diagonal elements of $K$ are positive, but in this case at least one element of $K$ is almost certain to be small relative to $|||T|||$. If there is any doubt as to whether or not $T$ is nearly singular, then the user should consider examining the diagonal elements of $K$.

9. Example

To solve the symmetric positive-definite equations

$$Tx = b$$

$$1 1$$

and

$$Tx = b$$

$$2 2$$

where

$$\begin{pmatrix} 4 & -2 & 0 & 0 & 0 \\ -2 & 10 & -6 & 0 & 0 \\ 0 & 0 & 15 & 25 & 8 \\ 0 & 0 & 0 & 8 & 5 \end{pmatrix}, \quad \begin{pmatrix} 6 \\ 9 \\ 14 \\ 7 \end{pmatrix}, \quad \begin{pmatrix} 10 \\ 4 \\ 2 \end{pmatrix}$$

$$T=(0, -6, 29, 15, 0), \quad b=(2), \quad b=(9).$$

$$\begin{pmatrix} 0 & 15 & 25 & 8 \\ 0 & 0 & 15 & 25 \\ 0 & 0 & 0 & 8 \end{pmatrix}, \quad \begin{pmatrix} 1 \end{pmatrix}, \quad \begin{pmatrix} 14 \end{pmatrix}, \quad \begin{pmatrix} 2 \end{pmatrix}$$

The equations are solved by two calls to F04FAF, the first with JOB = 0 and the second, using the factorization from the first call, with JOB = 1.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations

F04JGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

F04JGF finds the solution of a linear least-squares problem, \( Ax=b \), where \( A \) is a real \( m \) by \( n \) (\( m \geq n \)) matrix and \( b \) is an \( m \) element vector. If the matrix of observations is not of full rank, then the minimal least-squares solution is returned.

2. Specification

```fortran
SUBROUTINE F04JGF (M, N, A, NRA, B, TOL, SVD, SIGMA, 
                   1 IRANK, WORK, LWORK, IFAIL)
  INTEGER M, N, NRA, IRANK, LWORK, IFAIL
  DOUBLE PRECISION A(NRA,N), B(M), TOL, SIGMA, WORK(LWORK)
  LOGICAL SVD
```

3. Description

The minimal least-squares solution of the problem \( Ax=b \) is the vector \( x \) of minimum (Euclidean) length which minimizes the length of the residual vector \( r=b-Ax \).

The real \( m \) by \( n \) (\( m \geq n \)) matrix \( A \) is factorized as

\[
A = Q U
\]

where \( Q \) is an \( m \) by \( m \) orthogonal matrix and \( U \) is an \( n \) by \( n \) upper triangular matrix. If \( U \) is of full rank, then the least-squares solution is given by

\[
x = (U^T 0)Q b.
\]

If \( U \) is not of full rank, then the singular value decomposition of \( U \) is obtained so that \( U \) is factorized as

\[
U = RP^T,
\]

where \( R \) and \( P \) are \( n \) by \( n \) orthogonal matrices and \( D \) is the \( n \) by \( n \) diagonal matrix

\[
D = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n),
\]

with \( \sigma_k \geq \sigma_{k+1} \geq \ldots \geq \sigma_n \geq 0 \), these being the singular values of \( A \). If the singular values \( \sigma_1, \ldots, \sigma_k \) are not zero, then the minimal least-squares solution is given by

\[
x = (U^T 0)Q b.
\]
negligible, but \((\sigma)\) is not negligible, relative to the data errors in \(A\), then the rank of \(A\) is taken to be \(k\) and the minimal least-squares solution is given by

\[
\begin{pmatrix}
-1 \\
S & 0 & 0 \\
0 & R & 0 \\
0 & 0 & I
\end{pmatrix}
\begin{pmatrix}
T \\
S & 0 \\
0 & R \\
0 & 0 & I
\end{pmatrix}
\begin{pmatrix}
x \\ P \\ Q\ b,
\end{pmatrix}
\]

where \(S = \text{diag}((\sigma)_1, (\sigma)_2, \ldots, (\sigma)_k)\).

This routine obtains the factorizations by a call to F02WDF(*).

The routine also returns the value of the standard error

\[
\sqrt{r^T r}
\]

\[
= 0, \quad \text{if } m = k, \quad r^T r \text{ being the residual sum of squares and } k \text{ the rank of } A.
\]

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: \(m\), the number of rows of \(A\). Constraint: \(M \geq N\).

2: N -- INTEGER Input
   On entry: \(n\), the number of columns of \(A\). Constraint: \(1 \leq N \leq M\).

3: A(NRA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the \(m\) by \(n\) matrix \(A\). On exit: if SVD is returned as .FALSE., \(A\) is overwritten by details of the QU factorization of \(A\) (see F02WDF(*) for further details). If SVD is returned as .TRUE., the first \(n\) rows of \(A\) are overwritten by the right-hand singular vectors, stored by rows; and the remaining rows of the array are used as workspace.

4: NRA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the
(sub)program from which F04JGF is called.
Constraint: NRA >= M.

5: B(M) -- DOUBLE PRECISION array Input/Output
On entry: the right-hand side vector b. On exit: the first
n elements of B contain the minimal least-squares solution
vector x. The remaining m-n elements are used for workspace.

6: TOL -- DOUBLE PRECISION Input
On entry: a relative tolerance to be used to determine the
rank of A. TOL should be chosen as approximately the largest
relative error in the elements of A. For example, if the
elements of A are correct to about 4 significant figures
\(-4\)
then TOL should be set to about 5\times10^{-4}. See Section 8 for a
description of how TOL is used to determine rank. If TOL is
outside the range ((\text{epsilon}),1.0), where (\text{epsilon}) is the
machine precision, then the value (\text{epsilon}) is used in place
of TOL. For most problems this is unreasonably small.

7: SVD -- LOGICAL Output
On exit: SVD is returned as .FALSE. if the least-squares
solution has been obtained from the QU factorization of A.
In this case A is of full rank. SVD is returned as .TRUE. if
the least-squares solution has been obtained from the
singular value decomposition of A.

8: SIGMA -- DOUBLE PRECISION Output
On exit: the standard error, i.e., the value \(\sqrt{r \cdot r/(m-k)}\)
when m\neq k, and the value zero when m=k. Here r is the
residual vector b-Ax and k is the rank of A.

9: IRANK -- INTEGER Output
On exit: k, the rank of the matrix A. It should be noted
that it is possible for IRANK to be returned as n and SVD to
be returned as .TRUE.. This means that the matrix U only
just failed the test for non-singularity.

10: WORK(LWORK) -- DOUBLE PRECISION array Output
On exit: if SVD is returned as .FALSE., then the first n
elements of WORK contain information on the QU factorization
of A (see parameter A above and F02WDF(*)), and WORK(n+1)
\(-1\)
contains the condition number \( ||U|| \cdot ||U|| \) of the
\( E \)
upper triangular matrix U.

If SVD is returned as .TRUE., then the first n elements of WORK contain the singular values of A arranged in descending order and WORK(n+1) contains the total number of iterations taken by the QR algorithm. The rest of WORK is used as workspace.

11: LWORK -- INTEGER Input
On entry: the dimension of the array WORK as declared in the (sub)program from which F04JGF is called.
Constraint: LWORK >= 4*N.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry N < 1,
or M < N,
or NRA < M,
or LWORK < 4*N.

IFAIL= 2
The QR algorithm has failed to converge to the singular values in 50*N iterations. This failure can only happen when the singular value decomposition is employed, but even then it is not likely to occur.

7. Accuracy

The computed factors Q, U, R, D and P satisfy the relations

\[ (U) \quad (R \ 0) (D) \ T \]
\[ Q(0) = A + E , \ Q(0 \ I)(0) P = A + F , \]

where
\[ ||E|| \leq c(\text{epsilon}) ||A|| , \]
\[ \begin{array}{c}
\text{2 1} \\
\text{2 2}
\end{array} \]
\[ ||F|| \leq c(\text{epsilon}) ||A|| , \]
\[ \begin{array}{c}
\text{2 2} \\
\text{2 1}
\end{array} \]
(epsilon) being the machine precision, and c and c being modest functions of m and n. Note that \[ ||A|| = (\sigma) . \]

For a fuller discussion, covering the accuracy of the solution x see Lawson and Hanson [1], especially pp 50 and 95.

8. Further Comments

If the least-squares solution is obtained from the QU factorization then the time taken by the routine is approximately proportional to n (3m-n). If the least-squares solution is obtained from the singular value decomposition then the time taken is approximately proportional to n (3m+19n). The approximate proportionality factor is the same in each case.

This routine is column biased and so is suitable for use in paged environments.

Following the QU factorization of \( A \) the condition number

\[ c(U) = \frac{||U|| \cdot ||U|\rangle}{||E|| \cdot ||E||} \]

is determined and if \( c(U) \) is such that

\[ c(U) > \text{TOL} > 1.0 \]

then \( U \) is regarded as singular and the singular values of \( A \) are computed. If this test is not satisfied, \( U \) is regarded as non-singular and the rank of \( A \) is set to \( n \). When the singular values are computed the rank of \( A \), say \( k \), is returned as the largest integer such that

\[ (\sigma) > \text{TOL} \times (\sigma) , \]
\[ k \]

unless \( (\sigma) = 0 \) in which case \( k \) is returned as zero. That is,
singular values which satisfy \( \sigma_i \leq \text{TOL} \cdot \sigma_1 \) are regarded as negligible because relative perturbations of order \( \text{TOL} \) can make such singular values zero.

9. Example

To obtain a least-squares solution for \( r = b - Ax \), where

\[
\begin{align*}
A &= (1.75 & 1.75 & 0.35 & -0.35), \\
B &= (4)
\end{align*}
\]

and the value \( \text{TOL} \) is to be taken as \( 5 \times 10^{-4} \).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

### F04MAF

**F04MAF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details.

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

To solve a sparse symmetric positive-definite system of linear equations, \( Ax = b \), using a pre-conditioned conjugate gradient method, where \( A \) has been factorized by F01MAF.

2. **Specification**

```fortran
SUBROUTINE F04MAF (N, NZ, A, LICN, IRN, LIRN, ICN, B, ACC, 
                    NOITS, WKEEP, WORK, IKEEP, INFORM, 
                    IFAIL)
INTEGER N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN), 
        NOITS(2), IKEEP(2*N), INFORM(4), IFAIL
DOUBLE PRECISION A(LICN), B(N), ACC(2), WKEEP(3*N), WORK 
                    (3*N)
```
3. Description

F04MAF solves the $n$ linear equations

$$Ax=b,$$

(1)

where $A$ is a sparse symmetric positive-definite matrix, following the incomplete Cholesky factorization by F01MAF, given by

$$T T$$

$$C=PLDL P, \ WAW=C+E,$$

where $P$ is a permutation matrix, $L$ is a unit lower triangular matrix, $D$ is a diagonal matrix with positive diagonal elements, $E$ is an error matrix representing elements dropped during the factorization and diagonal elements that have been modified to ensure that $C$ is positive-definite, and $W$ is a diagonal matrix, chosen to make the diagonal elements of $WAW$ unity.

Equation (1) is solved by applying a pre-conditioned conjugate gradient method to the equations

$$-(WAW)(W x)=Wb,$$

(2)

using $C$ as the pre-conditioning matrix. Details of the conjugate gradient method are given in Munksgaard [1].

The iterative procedure is terminated if

$$\|Wr\| \leq (\eta),$$

(3)

where $r$ is the residual vector $r=b-Ax$, $\|r\|$ denotes the Euclidean length of $r$, $(\eta)$ is a user-supplied tolerance and $x$ is the current approximation to the solution. Notice that

$$-1$$

$$Wr=Wb-(WAW)(W x),$$

so that $Wr$ is the residual of the normalised equations (2).

F04MAF is based on the Harwell Library routine MA31B.

4. References

5. Parameters

1: N -- INTEGER  
   On entry: n, the order of the matrix A. Constraint: N >= 1.

2: NZ -- INTEGER  
   On entry: the number of non-zero elements in the upper  
   triangular part of the matrix A, including the number of  
   elements on the leading diagonal. Constraint: NZ >= N.

3: A(LICN) -- DOUBLE PRECISION array  
   On entry: the first LROW elements, where LROW is the value  
   supplied in INFORM(1), must contain details of the  
   factorization, as returned by F01MAF.

4: LICN -- INTEGER  
   On entry: the length of the array A, as declared in the  
   (sub)program from which F04MAF is called. It need never be  
   larger than the value of LICN supplied to F01MAF.  
   Constraint: LICN >= INFORM(1).

5: IRN(LIRN) -- INTEGER array  
   On entry: the first LCOL elements, where LCOL is the value  
   supplied in INFORM(2), must contain details of the  
   factorization, as returned by F01MAF.

6: LIRN -- INTEGER  
   On entry: the length of the array IRN, as declared in the  
   (sub)program from which F04MAF is called. It need never be  
   larger than the value of LIRN supplied to F01MAF.  
   Constraint: LIRN >= INFORM(2).

7: ICN(LICN) -- INTEGER array  
   On entry: the first LROW elements, where LROW is the value  
   supplied in INFORM(1), must contain details of the  
   factorization, as returned by F01MAF.

8: B(N) -- DOUBLE PRECISION array  
   On entry: the right-hand side vector b. On exit: B is  
   overwritten by the solution vector x.

9: ACC(2) -- DOUBLE PRECISION array  
   On entry: ACC(1) specifies the tolerance for convergence,  
   (eta), in equation (3) of Section 3. If ACC(1) is outside  
   the range [(epsilon),1], where (epsilon) is the machine  
   precision, then the value (epsilon) is used in place of ACC  
   (1). ACC(2) need not be set. On exit: ACC(2) contains the  
   actual value of ||Wr|| at the final point. ACC(1) is

2
unchanged.

10: NOITS(2) -- INTEGER array Input/Output
On entry: NOITS(1) specifies the maximum permitted number of
iterations. If NOITS(1) < 1, then the value 100 is used in
its place. NOITS(2) need not be set. On exit: NOITS(2)
contains the number of iterations taken to converge. NOITS
(1) is unchanged.

11: WKEEP(3*N) -- DOUBLE PRECISION array Input
On entry: WKEEP must be unchanged from the previous call of
F01MAF.

12: WORK(3*N) -- DOUBLE PRECISION array Output
On exit: WORK(1) contains a lower bound for the condition
number of A. The rest of the array is used for workspace.

13: IKEEP(2*N) -- INTEGER array Input
On entry: IKEEP must be unchanged from the previous call of
F01MAF.

14: INFORM(4) -- INTEGER array Input
On entry: INFORM must be unchanged from the previous call of
F01MAF.

15: IFAIL -- INTEGER Input/Output
For this routine, the normal use of IFAIL is extended to
control the printing of error and warning messages as well
as specifying hard or soft failure (see the Essential
Introduction).

Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a
must have a value of 0 or 1.

a=0 specifies hard failure, otherwise soft failure;

b=0 suppresses error messages, otherwise error messages
will be printed (see Section 6);

C=0 suppresses warning messages, otherwise warning
messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e.,
hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL
contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:
For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL= 1
On entry N < 1,
or NZ < N,
or LICN < INFORM(1),
or LIRN < INFORM(2).

IFAIL= 2
Convergence has not taken place within the requested NOITS (1) number of iterations. ACC(2) gives the value \( ||W_r|| \),
\[ 2 \]
for the final point. Either too few iterations have been allowed, or the requested convergence criterion cannot be met.

IFAIL= 3
The matrix A is singular, or nearly singular. Singularity has been detected during the conjugate gradient iterations, so that the computations are not complete.

IFAIL= 4
The matrix A is singular, or nearly singular. The message output on the current error message channel will include an estimate of the condition number of A. In the case of soft failure an approximate solution is returned such that the value \( ||W_r|| \) is given by ACC(2) and the estimate (a lower bound) of the condition number is returned in WORK(1).

7. Accuracy
On successful return, or on return with IFAIL = 2 or IFAIL = 4 the computed solution will satisfy equation (3) of Section 3, with (eta) = ACC(2).

8. Further Comments
The time taken by the routine will depend upon the sparsity of the factorization and the number of iterations required. The number of iterations will be affected by the nature of the factorization supplied by F01MAF. The more incomplete the factorization, the higher the number of iterations required by F04MAF.
When the solution of several systems of equations, all with the same matrix of coefficients, $A$, is required, then F01MAF need be called only once to factorize $A$. This is illustrated in the context of an eigenvalue problem in the example program for F02FJF.

9. Example

The example program illustrates the use of F01MAF in conjunction with F04MAF to solve the 16 linear equations $Ax=b$, where

\[
\begin{pmatrix}
1 & a & a \\
a & 1 & a & a \\
a & a & 1 & a \\
a & a & a & 1 \\
a & a & a & a \\
a & a & a & 1 \\
a & a & a & a \\
a & a & a & 1 \\
\end{pmatrix}
\]

where $a=-\frac{1}{4}$.

\[
T \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{pmatrix}
\]

\[
b = \begin{pmatrix}
\end{pmatrix}
\]

The $n$ by $n$ matrix $A$ arises in the solution of Laplace’s equation in a unit-square, using a five-point formula with a 6 by 6 discretisation, with unity on the boundaries.

The drop tolerance, DROPTL, is taken as 0.1 and the density factor, DENS, is taken as 0.8. The value IFAIL = 111 is used so that advisory and error messages will be printed, but soft failure would occur if IFAIL were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of routines
F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations

F04MBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04MBF solves a system of real sparse symmetric linear equations using a Lanczos algorithm.

2. Specification

```fortran
SUBROUTINE F04MBF (N, B, X, APROD, MSOLVE, PRECON, SHIFT, RTOL, ITNLIM, MSGLVL, ITN, ANORM, ACOND, RNORM, XNORM, WORK, RWORK, LRWORK, IWORK, LIWORK, INFORM, IFAIL)
INTEGER N, ITNLIM, MSGLVL, ITN, LRWORK, IWORK
DOUBLE PRECISION B(N), X(N), SHIFT, RTOL, ANORM, ACOND, RNORM, XNORM, WORK(N,5), RWORK(LRWORK)
LOGICAL PRECON
EXTERNAL APROD, MSOLVE
```

3. Description

F04MBF solves the system of linear equations

\[(A-(\lambda)I)x=b\]  \hspace{1cm} (3.1)

where \(A\) is an \(n\) by \(n\) sparse symmetric matrix and \(\lambda\) is a scalar, which is of course zero if the solution of the equations

\[Ax=b\]

is required. It should be noted that neither \(A\) nor \((A-(\lambda)I)\) need be positive-definite.

\(\lambda\) is supplied as the parameter \(\text{SHIFT}\), and allows F04MBF to be used for finding eigenvectors of \(A\) in methods such as Rayleigh...
quotient iteration (see for example Lewis [1]), in which case
(lamba) will be an approximation to an eigenvalue of A and b an
approximation to an eigenvector of A.

The routine also provides an option to allow pre-conditioning and
this will often reduce the number of iterations required by
F04MBF.

F04MBF is based upon algorithm SYMMLQ (see Paige and Saunders
[2]) and solves the equations by an algorithm based upon the

Lanczos process. Details of the method are given in Paige and
Saunders [2]. The routine does not require A explicitly, but A is
specified via a user-supplied routine APROD which, given an n
element vector c, must return the vector z given by

\[ z = Ac. \]

The pre-conditioning option is based on the following reasoning.
If A can be expressed in the form

\[ A = I + B \]

where B is of rank (rho), then the Lanczos process converges (in
exact arithmetic) in at most (rho) iterations. If more generally
A can be expressed in the form

\[ A = M + C \]

where M is symmetric positive-definite and C has rank (rho), then

\[
\begin{pmatrix}
-(1/2) & -(1/2) & -(1/2) & -(1/2) \\
M & AM & = & I + M & CM
\end{pmatrix}
\]

\[-(1/2) & -(1/2) \]

and M AM also has rank (rho), and the Lanczos process

\[-(1/2) & -(1/2) \]

applied to M AM would again converge in at most (rho)
iterations. On a computer, the number of iterations may be
greater than (rho), but the Lanczos process may still be expected
\[-(1/2) & -(1/2) \]
to converge rapidly. F04MBF does not require M AM to
be formed explicitly, but implicitly solves the equations

\[
\begin{pmatrix}
-(1/2) & -(1/2) & -(1/2) & 1/2 \\
M & (A-(lambda)I) & M & y=M & b, y=M & x
\end{pmatrix}
\]

with the user being required to supply a routine MSOLVE to solve
the equations
\( Mz = c. \) \hspace{1cm} (3.3)

For the pre-conditioning option to be effective, it is desirable that equations (3.3) can be solved efficiently. The example program in Section 9 illustrates the use of this option.

If we let \( r \) denote the residual vector
\[
r = b - (A - (\lambda)I)x
\]
corresponding to an iterate \( x \), then, when pre-conditioning has not been requested, the iterative procedure is terminated if it is estimated that
\[
||r|| \leq \text{tol}.||A - (\lambda)I||.||x||, \hspace{1cm} (3.4)
\]
where \( \text{tol} \) is a user-supplied tolerance, \( ||r|| \) denotes the Euclidean length of the vector \( r \) and \( ||A|| \) denotes the Frobenius (Euclidean) norm of the matrix \( A \). When pre-conditioning has been requested, the iterative procedure is terminated if it is estimated that
\[
||M r|| \leq \text{tol}.||M (A - (\lambda)I)M||.||M x||. \hspace{1cm} (3.5)
\]
Note that
\[
M r = (M b) - M (A - (\lambda)I)M (M x)
\]
so that \( M r \) is the residual vector corresponding to equation (3.2). The routine will also terminate if it is estimated that
\[
||A - (\lambda)I||.||x|| > ||b||/(\epsilon), \hspace{1cm} (3.6)
\]
where \( \epsilon \) is the machine precision, when pre-conditioning has not been requested; or if it is estimated that
\[
||M (A - (\lambda)I)M||.||M x|| > ||M b||/(\epsilon)
\]
when pre-conditioning has been requested. If (3.6) is satisfied then \( x \) is almost certainly an eigenvector of \( A \) corresponding to the eigenvalue \( \lambda \). If \( \lambda \) was set to 0 (for the solution of \( Ax = b \)), then this condition simply means that \( A \) is effectively singular.
4. References


5. Parameters

1: $N$ -- INTEGER Input
   On entry: $n$, the order of the matrix $A$. Constraint: $N \geq 1$.

2: $B(N)$ -- DOUBLE PRECISION array Input
   On entry: the right-hand side vector $b$.

3: $X(N)$ -- DOUBLE PRECISION array Output
   On exit: the solution vector $x$.

4: $APROD$ -- SUBROUTINE, supplied by the user.
   External Procedure
   $APROD$ must return the vector $y=Ax$ for a given vector $x$.

   Its specification is:

   
   \begin{verbatim}
   SUBROUTINE APROD (IFLAG, N, X, Y, RWORK, LRWORK, IWORK, LIWORK)
   INTEGER IFLAG, N, LRWORK, LIWORK, IWORK
   DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)
   
   1: IFLAG -- INTEGER Input/Output
      On entry: $IFLAG$ is always non-negative. On exit: $IFLAG$
      may be used as a flag to indicate a failure in the
      computation of $Ax$. If $IFLAG$ is negative on exit from
      $APROD$, F04MBF will exit immediately with $IFAIL$ set to
      $IFLAG$.
   
   2: $N$ -- INTEGER Input
      On entry: $n$, the order of the matrix $A$.
   
   3: $X(N)$ -- DOUBLE PRECISION array Input
      On entry: the vector $x$ for which $Ax$ is required.
   
   4: $Y(N)$ -- DOUBLE PRECISION array Output
      On exit: the vector $y=Ax$.
   \end{verbatim}
APROD is called from F04MBF with the parameters RWORK, LRWORK, IWORK and LIWORK as supplied to F04MBF. The user is free to use the arrays RWORK and IWORK to supply information to APROD and MSOLVE as an alternative to using COMMON.

APROD must be declared as EXTERNAL in the (sub)program from which F04MBF is called. Parameters denoted as Input must not be changed by this procedure.

MSOLVE -- SUBROUTINE, supplied by the user.

External Procedure

MSOLVE is only referenced when PRECON is supplied as .TRUE.. When PRECON is supplied as .FALSE., then F04MBF may be called with APROD as the actual argument for MSOLVE. When PRECON is supplied as .TRUE., then MSOLVE must return the solution y of the equations My=x for a given vector x, where M must be symmetric positive-definite.

Its specification is:

```
SUBROUTINE MSOLVE (IFLAG, N, X, Y, RWORK, LRWORK, IWORK, LIWORK)
INTEGER IFLAG, N, LRWORK, IWORK, LIWORK
DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)

1: IFLAG -- INTEGER Input/Output
On entry: IFLAG is always non-negative. On exit: IFLAG may be used as a flag to indicate a failure in the solution of My=x.

If IFLAG is negative on exit from MSOLVE, F04MBF will exit immediately with IFAIL set to IFLAG.

2: N -- INTEGER Input
On entry: n, the order of the matrix M.

3: X(N) -- DOUBLE PRECISION array Input
On entry: the vector x for which the equations My=x are to be solved.

4: Y(N) -- DOUBLE PRECISION array Output
On exit: the solution to the equations My=x.
```
5: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace

6: LRWORK -- INTEGER Input

7: IWORK(LIWORK) -- INTEGER array User Workspace

8: LIWORK -- INTEGER Input

MSOLVE is called from F04MBF with the parameters RWORK, LRWORK, IWORK and LIWORK as supplied to F04MBF. The user is free to use the arrays RWORK and IWORK to supply information to APROD and MSOLVE as an alternative to using COMMON.

MSOLVE must be declared as EXTERNAL in the (sub)program from which F04MBF is called. Parameters denoted as Input must not be changed by this procedure.

6: PRECON -- LOGICAL Input

On entry: PRECON specifies whether or not pre-conditioning is required. If PRECON = .TRUE., then pre-conditioning will be invoked and MSOLVE will be referenced by F04MBF; if PRECON = .FALSE., then MSOLVE is not referenced.

7: SHIFT -- DOUBLE PRECISION Input

On entry: the value of (lambda). If the equations Ax=b are to be solved, then SHIFT must be supplied as zero.

8: RTOL -- DOUBLE PRECISION Input

On entry: the tolerance for convergence, tol, of equation (3.4). RTOL should not normally be less than (epsilon), where (epsilon) is the machine precision.

9: ITNLIM -- INTEGER Input

On entry: an upper limit on the number of iterations. If ITNLIM <= 0, then the value N is used in place of ITNLIM.

10: MSG_LVL -- INTEGER Input

On entry: the level of printing from F04MBF. If MSG_LVL <= 0, then no printing occurs, but otherwise messages will be output on the advisory message channel (see X04ABF). A description of the printed output is given in Section 5.1 below. The level of printing is determined as follows:

MSG_LVL = 0
No printing.

MSG_LVL = 1
A brief summary is printed just prior to return from F04MBF.

MSG_LVL >= 2
A summary line is printed periodically to monitor the progress of F04MBF, together with a brief summary just prior to return from F04MBF.

11: ITN -- INTEGER Output
On exit: the number of iterations performed.

12: ANORM -- DOUBLE PRECISION Output
On exit: an estimate of $||A-(\lambda)I||$ when PRECON = -(1/2) -(1/2) .FALSE., and $||M^{-1/2} (A-(\lambda)I)M||$ when PRECON = .TRUE.. This will usually be a substantial under-estimate.

13: ACOND -- DOUBLE PRECISION Output
On exit: an estimate of the condition number of $(A-(\lambda)I)$ when PRECON = .FALSE., and of $-(1/2) -(1/2) M^{-1/2} (A-(\lambda)I)M$ when PRECON = .TRUE.. This will usually be a substantial under-estimate.

14: RNORM -- DOUBLE PRECISION Output
On exit: $||r||$, where $r=b-(A-(\lambda)I)x$ and $x$ is the solution returned in X.

15: XNORM -- DOUBLE PRECISION Output
On exit: $||x||$, where $x$ is the solution returned in X.

16: WORK(5*N) -- DOUBLE PRECISION array Workflow

17: RWORK(LRWORK) -- DOUBLE PRECISION array User Workflow
RWORK is not used by F04MBF, but is passed directly to routines APROD and MSOLVE and may be used to pass information to these routines.

18: LRWORK -- INTEGER Input
On entry: the length of the array RWORK as declared in the (sub)program from which F04MBF is called. Constraint: LRWORK >= 1.

19: IWORK(LIWORK) -- INTEGER array User Workspace
IWORK is not used by F04MBF, but is passed directly to routines APROD and MSOLVE and may be used to pass information to these routines.

20: LIWORK -- INTEGER Input
On entry: the length of the array IWORK as declared in the (sub)program from which F04MBF is called. Constraint: LIWORK >= 1.

21: INFORM -- INTEGER Output
On exit: the reason for termination of F04MBF as follows:

INFORM = 0
The right-hand side vector b=0 so that the exact solution is x=0. No iterations are performed in this case.

INFORM = 1
The termination criterion of equation (3.4) has been satisfied with tol as the value supplied in RTOL.

INFORM = 2
The termination criterion of equation (3.4) has been satisfied with tol equal to (epsilon), where (epsilon) is the machine precision. The value supplied in RTOL must have been less than (epsilon) and was too small for the machine.

INFORM = 3
The termination criterion of equation (3.5) has been satisfied so that X is almost certainly an eigenvector of A corresponding to the eigenvalue SHIFT.

The values INFORM = 4 and INFORM = 5 correspond to failure with IFAIL = 3 or IFAIL = 2 respectively (see Section 6) and when IFAIL is negative, INFORM will be set to the same negative value.

22: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

5.1. Description of the Printed Output

When MSGVL > 0, then F04MBF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF).

The following notation is used in the output.

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(1/2)</td>
<td>RBAR M</td>
</tr>
<tr>
<td></td>
<td>(b-(A-(lambda)I)x)=r</td>
</tr>
<tr>
<td>-(1/2)</td>
<td>ABAR M</td>
</tr>
<tr>
<td></td>
<td>(A-(lambda)I)M =A</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\frac{1}{2} & \\
Y & M x \\
R & b-(A-(\lambda)I)x \\
\text{NORM}(A) & ||A||
\end{align*}
\]

Of course, when pre-conditioning has not been requested then the first three reduce to \((b-(A-(\lambda)I)x)\), \((A-(\lambda)I)\) and \(x\) respectively. When MSGVL >= 2 then some initial information is printed and the following notation is used.

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>T^-1 1/2</td>
<td>BETA1 ((b M b) == (beta))</td>
</tr>
<tr>
<td>2 -(1/2) T -(1/2) -(1/2)</td>
<td>ALFA1 ((1/(beta)) (M b) (M AM) (M b) == (alpha))</td>
</tr>
</tbody>
</table>

and a summary line is printed periodically giving the following information:

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITN</td>
<td>Iteration number, (k).</td>
</tr>
<tr>
<td>L</td>
<td>(x_1(LQ)) The first element of the vector (x), where (x) is the (k) (k) current iterate. See Paige and Saunders [2] for details.</td>
</tr>
<tr>
<td>C</td>
<td>(x_1(CG)) The first element of the vector (x), where (x) is the (k) (k) vector that would be obtained by conjugate gradients. See Paige and Saunders [2] for details.</td>
</tr>
</tbody>
</table>

\text{NORM(RBAR)} \(||r||\), where \(r\) is as defined above and \(x\) is either \(L\) \(C\) \(x\) or \(x\) depending upon which is the best current \(k\) \(k\) approximation to the solution. (See LQ/CG below).
NORM(T) The value $\|T\|_k$, where $T$ is the tridiagonal $k \times k$ matrix of the Lanczos process. This increases monotonically and is a lower bound on $\|A\|$.

COND(L) A monotonically increasing lower bound on the $\|A^{-1}\| L$ condition number of $A$, $\|A\| \|L\|$.

LQ/CG $L$ is printed if $x$ is the best current $k$ approximation to the solution and $C$ is printed otherwise.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL < 0
A negative value of IFAIL indicates an exit from F04MBF because the user has set IFLAG negative in APROD or MSOLVE. The value of IFAIL will be the same as the user’s setting of IFLAG.

IFAIL = 1
On entry $N < 1$,

or $LRWORK < 1$,

or $LIWORK < 1$.

IFAIL = 2
The pre-conditioning matrix $M$ does not appear to be positive-definite. The user should check that MSOLVE is working correctly.

IFAIL = 3
The limit on the number of iterations has been reached. If IFAIL = 1 on entry then the latest approximation to the solution is returned in $X$ and the values ANORM, ACOND, RNORM and XNORM are also returned.

The value of INFORM contains additional information about the termination of the routine and users must examine INFORM to judge whether the routine has performed successfully for the problem in hand. In particular INFORM = 3 denotes that the matrix $A-(\lambda)I$ is effectively singular: if the purpose of calling
F04MBF is to solve a system of equations $Ax=b$, then this condition must be regarded as a failure, but if the purpose is to compute an eigenvector, this result would be very satisfactory.

7. Accuracy

The computed solution $x$ will satisfy the equation

$$r = b - (A - (\lambda)I)x$$

where the value $||r||$ is as returned in the parameter RNORM.

8. Further Comments

The time taken by the routine is likely to be principally determined by the time taken in APROD and, when pre-conditioning has been requested, in MSOLVE. Each of these routines is called once every iteration.

The time taken by the remaining operations in F04MBF is approximately proportional to $n$.

9. Example

To solve the 10 equations $Ax=b$ given by

$$
\begin{pmatrix}
2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\
\end{pmatrix} \begin{pmatrix}
6 \\
4 \\
4 \\
4 \\
4 \\
4 \\
4 \\
4 \\
4 \\
4 \\
6 \\
\end{pmatrix}
$$

A=(0 0 0 0 1 2 1 0 0 0), \ b=(4).

The tridiagonal part of $A$ is positive-definite and such tridiagonal equations can be solved efficiently by F04FAF. The form of $A$ suggests that this tridiagonal part is a good candidate for the pre-conditioning matrix $M$ and so we illustrate the use of F04MBF by pre-conditioning with the 10 by 10 matrix

$$
\begin{pmatrix}
2 & 1 & 0 & \ldots & 0 \\
1 & 2 & 1 & \ldots & 0 \\
0 & 1 & 2 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & \ldots & 2 \\
\end{pmatrix}
$$
Since $A-M$ has only 2 non-zero elements and is obviously of rank 2, we can expect F04MBF to converge very quickly in this example. Of course, in practical problems we shall not usually be able to make such a good choice of $M$.

The example sets the tolerance $\text{RTOL} = 10^{-5}$.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**F04 -- Simultaneous Linear Equations**

**F04MCF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

F04MCF computes the approximate solution of a system of real linear equations with multiple right-hand sides, $AX=B$, where $A$ is a symmetric positive-definite variable-bandwidth matrix, which has previously been factorized by F01MCF. Related systems may also be solved.

2. **Specification**

```fortran
SUBROUTINE F04MCF (N, AL, LAL, D, NROW, IR, B, NRB, ISELCT, X, NRX, IFAIL)
    INTEGER N, LAL, NROW(N), IR, NRB, ISELCT, NRX, IFAIL
    DOUBLE PRECISION AL(LAL), D(N), B(NRB,IR), X(NRX,IR)
```

3. **Description**

The normal use of this routine is the solution of the systems $AX=B$, following a call of F01MCF to determine the Cholesky $T$ factorization $A=LDL^T$ of the symmetric positive-definite variable-bandwidth matrix $A$.

However, the routine may be used to solve any one of the following systems of linear algebraic equations:
\[ T \]

(1) \( LDL X = B \) (usual system),

(2) \( LDX = B \) (lower triangular system),

(3) \( DLX = B \) (upper triangular system),

(4) \( LLX = B \)

(5) \( LX = B \) (unit lower triangular system),

(6) \( LX = B \) (unit upper triangular system).

\( L \) denotes a unit lower triangular variable-bandwidth matrix of order \( n \), \( D \) a diagonal matrix of order \( n \), and \( B \) a set of right-hand sides.

The matrix \( L \) is represented by the elements lying within its envelope i.e., between the first non-zero of each row and the diagonal (see Section 9 for an example). The width \( \text{NROW}(i) \) of the \( i \)th row is the number of elements between the first non-zero element and the element on the diagonal inclusive.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
   On entry: \( n \), the order of the matrix \( L \). Constraint: \( N \geq 1 \).

2: \( AL(LAL) \) -- DOUBLE PRECISION array Input
   On entry: the elements within the envelope of the lower triangular matrix \( L \), taken in row by row order, as returned by \( \text{F01MCF} \). The unit diagonal elements of \( L \) must be stored explicitly.

3: \( LAL \) -- INTEGER Input
   On entry: the dimension of the array \( AL \) as declared in the (sub)program from which \( \text{F04MCF} \) is called.
   Constraint: \( LAL \geq \text{NROW}(1) + \text{NROW}(2) + \ldots + \text{NROW}(n) \).

4: \( D(N) \) -- DOUBLE PRECISION array Input
On entry: the diagonal elements of the diagonal matrix \( D \). \( D \) is not referenced if \( ISELCT \geq 4 \).

5: \( \text{NROW}(N) \) -- INTEGER array  
   Input  
   On entry: \( \text{NROW}(i) \) must contain the width of row \( i \) of \( L \),  
   i.e., the number of elements between the first (leftmost)  
   non-zero element and the element on the diagonal, inclusive.  
   Constraint: \( 1 \leq \text{NROW}(i) \leq i \).

6: \( IR \) -- INTEGER  
   Input  
   On entry: \( r \), the number of right-hand sides. Constraint: \( IR \geq 1 \).

7: \( B(NRB,IR) \) -- DOUBLE PRECISION array  
   Input  
   On entry: the \( n \) by \( r \) right-hand side matrix \( B \). See also  
   Section 8.

8: \( NRB \) -- INTEGER  
   Input  
   On entry:  
   the first dimension of the array \( B \) as declared in the  
   (sub)program from which F04MCF is called.  
   Constraint: \( NRB \geq N \).

9: \( ISELCT \) -- INTEGER  
   Input  
   On entry: \( ISELCT \) must specify the type of system to be  
   solved, as follows:

   \[
   \begin{align*}
   & T \\
   & ISELCT = 1: \text{solve } LDL \ X = B, \\
   & ISELCT = 2: \text{solve } LDX = B, \\
   & T \\
   & ISELCT = 3: \text{solve } DL \ X = B, \\
   & T \\
   & ISELCT = 4: \text{solve } LL \ X = B, \\
   & ISELCT = 5: \text{solve } LX = B, \\
   & T \\
   & ISELCT = 6: \text{solve } L \ X = B.
   \end{align*}
   \]

10: \( X(NRX,IR) \) -- DOUBLE PRECISION array  
    Output  
    On exit: the \( n \) by \( r \) solution matrix \( X \). See also Section 8.

11: \( NRX \) -- INTEGER  
    Input  
    On entry:  
    the first dimension of the array \( X \) as declared in the  
    (sub)program from which F04MCF is called.
Constraint: NRX >= N.

12: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry N < 1,
   or for some i, NROW(i)<1 or NROW(i) > i,
   or LAL < NROW(1) + NROW(2) +... + NROW(N).

IFAIL= 2
   On entry IR < 1,
   or NRB < N,
   or NRX < N.

IFAIL= 3
   On entry ISELCT < 1,
   or ISELCT > 6.

IFAIL= 4
   The diagonal matrix D is singular, i.e., at least one of the
   elements of D is zero. This can only occur if ISELCT <= 3.

IFAIL= 5
   At least one of the diagonal elements of L is not equal to
   unity.

7. Accuracy

The usual backward error analysis of the solution of triangular
system applies: each computed solution vector is exact for
slightly perturbed matrices L and D, as appropriate (cf.

8. Further Comments
The time taken by the routine is approximately proportional to
pr, where

\[ p = \text{NROW}(1) + \text{NROW}(2) + \ldots + \text{NROW}(n). \]

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for the parameters B and X, in which case the
solution matrix will overwrite the right-hand side matrix.
However this is not standard Fortran 77 and may not work in all
implementations.

9. Example

To solve the system of equations \( AX = B \), where

\[
\begin{align*}
(1 & 2 & 0 & 0 & 5 & 0) \\
(2 & 5 & 3 & 0 & 14 & 0) \\
(0 & 3 & 13 & 0 & 18 & 0) \\
A &= (0 & 0 & 0 & 16 & 8 & 24) \\
(5 & 14 & 18 & 0 & 55 & 17) \\
(0 & 0 & 0 & 24 & 17 & 77)
\end{align*}
\]

and

\[
\begin{align*}
(6 & -10) \\
(15 & -21) \\
(11 & -3) \\
B &= (0 & 24) \\
(51 & -39) \\
(46 & 67)
\end{align*}
\]

Here \( A \) is symmetric and positive-definite and must first be
factorized by \( \text{F01MCF} \).

The example program is not reproduced here. The source code for
each example program is distributed with the NAG Foundation
Library software and should be available on-line.
1. Purpose

F04QAF solves sparse unsymmetric equations, sparse linear least-squares problems and sparse damped linear least-squares problems, using a Lanczos algorithm.

2. Specification

```fortran
SUBROUTINE F04QAF (M, N, B, X, SE, APROD, DAMP, ATOL,
  1 BTOL, CONLIM, ITNLIM, MSGVL, ITN,
  2 ANORM, ACOND, RNORM, ARNORM, XNORM,
  3 WORK, RWORK, LRWORK, IWORK, LIWORK,
  4 INFORM, IFAIL)
  INTEGER M, N, ITNLIM, MSGLVL, ITN, LRWORK, IWORK
  DOUBLE PRECISION B(M), X(N), SE(N), DAMP, ATOL, BTOL,
  1 CONLIM, ANORM, ACOND, RNORM, ARNORM,
  2 XNORM, WORK(N,2), RWORK(LRWORK)
  EXTERNAL APROD
```

3. Description

F04QAF can be used to solve a system of linear equations

\[ Ax = b \]  \hspace{1cm} (3.1)

where A is an n by n sparse unsymmetric matrix, or can be used to solve linear least-squares problems, so that F04QAF minimizes the value (\(\rho\)) given by

\[ (\rho) = ||r||, \quad r = b - Ax \]  \hspace{1cm} (3.2)

where A is an m by n sparse matrix and ||r|| denotes the Euclidean length of r so that \(||r|| = r^T r\). A damping parameter, \(\lambda\), may be included in the least-squares problem in which case F04QAF minimizes the value (\(\rho\)) given by

\[ (\rho) = ||r||^2 + (\lambda) ||x||^2 \]  \hspace{1cm} (3.3)

(\(\lambda\)) is supplied as the parameter DAMP and should of course be zero if the solution to problems (3.1) or (3.2) is required. Minimizing (\(\rho\)) in (3.3) is often called ridge regression.

F04QAF is based upon algorithm LSQR (see Paige and Saunders [1] and [2]) and solves the problems by an algorithm based upon the Lanczos process. Details of the method are given in [1]. The routine does not require A explicitly, but A is specified via a user-supplied routine APROD which must perform the operations (.
y+Ax) and (x+A y) for a given n element vector x and m element vector y. A parameter to APROD specifies which of the two operations is required on a given entry.

The routine also returns estimates of the standard errors of the sample regression coefficients \( \hat{x}_i \), for \( i=1,2,\ldots,n \) given by the diagonal elements of the estimated variance-covariance matrix \( V \).

When problem (3.2) is being solved and \( A \) is of full rank, then \( V \) is given by

\[
V = s^2 (A^T A)^{-1}, \quad s = \frac{\rho}{m-n}, \quad m>n
\]

and when problem (3.3) is being solved then \( V \) is given by

\[
V = s^2 (A^T A + (\lambda)^2 I)^{-1}, \quad s = \frac{\rho}{m}, \quad \lambda = 0.
\]

Let \( A \) denote the matrix

\[
A = (A, \lambda = 0) ; A = (\lambda I), \lambda = 0,
\]

let \( r \) denote the residual vector

\[
r = r, \lambda = 0 ; r = (0) - Ax, \lambda = 0
\]

corresponding to an iterate \( x \), so that \( \rho = ||r|| \) is the function being minimized, and let \( ||A|| \) denote the Frobenius (Euclidean) norm of \( A \). Then the routine accepts \( x \) as a solution if it is estimated that one of the following two conditions is satisfied:

\[
\rho \leq tol \ ||A|| \ ||x|| + tol \ ||b||
\]

\[
||A^T r|| \leq tol \ ||A|| \rho
\]
where tol and tol are user-supplied tolerances which estimate the relative errors in A and b respectively. Condition (3.6) is appropriate for compatible problems where, in theory, we expect the residual to be zero and will be satisfied by an acceptable solution x to a compatible problem. Condition (3.7) is appropriate for incompatible systems where we do not expect the residual to be zero and is based upon the observation that, in theory,

\[ \text{T} \ A \ x = 0 \]

when x is a solution to the least-squares problem, and so (3.7) will be satisfied by an acceptable solution x to a linear least-squares problem.

The routine also includes a test to prevent convergence to solutions, x, with unacceptably large elements. This can happen if A is nearly singular or is nearly rank deficient. If we let the singular values of A be

\[ (\sigma_1) \geq (\sigma_2) \geq \ldots \geq (\sigma_n) \geq 0 \]

then the condition number of A is defined as

\[ \text{cond}(A) = \frac{(\sigma_1)}{(\sigma_k)} \]

where (\sigma_1) is the smallest non-zero singular value of A and k hence k is the rank of A. When k<n, then A is rank deficient, the least-squares solution is not unique and F04QAF will normally converge to the minimal length solution. In practice A will not have exactly zero singular values, but may instead have small singular values that we wish to regard as zero.
The routine provides for this possibility by terminating if

\[
\text{cond}(A) \geq c \quad \text{lim}
\]

where \( c \) is a user-supplied limit on the condition number of \( A \).

For problem (3.1) termination with this condition indicates that \( A \) is nearly singular and for problem (3.2) indicates that \( A \) is nearly rank deficient and so has near linear dependencies in its \( T \) columns. In this case inspection of \( ||r||, ||Ar|| \) and \( ||x|| \), which are all returned by the routine, will indicate whether or not an acceptable solution has been found. Condition (3.8), perhaps in conjunction with \((\lambda)=0\), can be used to try and 'regularise' least-squares solutions. A full discussion of the stopping criteria is given in Section 6 of reference Paige and Saunders [1].

Introduction of a non-zero damping parameter \( (\lambda) \) tends to reduce the size of the computed solution and to make its components less sensitive to changes in the data, and F04QAF is applicable when a value of \( (\lambda) \) is known a priori. To have an effect, \( (\lambda) \) should normally be at least \( \sqrt{(\epsilon)||A||} \) where \( (\epsilon) \) is the machine precision. For further discussion see Paige and Saunders [2] and the references given there.

Whenever possible the matrix \( A \) should be scaled so that the relative errors in the elements of \( A \) are all of comparable size. Such a scaling helps to prevent the least-squares problem from being unnecessarily sensitive to data errors and will normally reduce the number of iterations required. At the very least, in the absence of better information, the columns of \( A \) should be scaled to have roughly equal column length.

4. References


5. Parameters

1: M -- INTEGER  
   On entry: m, the number of rows of the matrix A.
   Constraint: M >= 1.

2: N -- INTEGER  
   On entry: n, the number of columns of the matrix A.
   Constraint: N >= 1.

3: B(M) -- DOUBLE PRECISION array  
   On entry: the right-hand side vector b. On exit: the array
   is overwritten.

4: X(N) -- DOUBLE PRECISION array  
   On exit: the solution vector x.

5: SE(N) -- DOUBLE PRECISION array  
   On exit: the estimates of the standard errors of the
   components of x. Thus SE(i) contains an estimate of the
   element v of the estimated variance-covariance matrix V.
   The estimates returned in SE will be the lower bounds on the
   actual estimated standard errors, but will usually have at
   least one correct figure.

6: APROD -- SUBROUTINE, supplied by the user.  
   External Procedure

   APROD must perform the operations y:=y+Ax and x:=x+A y for
   given vectors x and y.

   Its specification is:

   SUBROUTINE APROD (MODE, M, N, X, Y, RWORK,  
   1 LRWORK, IWORK, LIWORK)
   INTEGER MODE, M, N, LRWORK, LIWORK,  
   1 IWORK(LIWORK)
   DOUBLE PRECISION X(N), Y(M), RWORK(LRWORK)

   1: MODE -- INTEGER  
      On entry: MODE specifies which operation is to be
      performed:
      If MODE = 1, then APROD must compute y+Ax.

      T
      If MODE = 2, then APROD must compute x+A y.
   On exit: MODE may be used as a flag to indicate a
failure in the computation of $y+Ax$ or $x+Ay$. If MODE is negative on exit from APROD, F04QAF will exit immediately with IFAIL set to MODE.

2: M -- INTEGER Input
On entry: $m$, the number of rows of $A$.

3: N -- INTEGER Input
On entry: $n$, the number of columns of $A$.

4: X(N) -- DOUBLE PRECISION array Input/Output
On entry: the vector $x$. On exit: if MODE = 1, $X$ must be unchanged;
   If MODE = 2, $X$ must contain $x+Ay$.

5: Y(M) -- DOUBLE PRECISION array Input/Output
On entry: the vector $y$. On exit: if MODE = 1, $Y$ must contain $y+Ax$;
   If MODE = 2, $Y$ must be unchanged.

6: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace

7: LRWORK -- INTEGER Input

8: IWORK(LIWORK) -- INTEGER array User Workspace

APROD must be declared as EXTERNAL in the (sub)program from which F04QAF is called. Parameters denoted as Input must not be changed by this procedure.

7: DAMP -- DOUBLE PRECISION Input
On entry: the value $(\lambda)$. If either problem (3.1) or problem (3.2) is to be solved, then DAMP must be supplied as zero.

8: ATOL -- DOUBLE PRECISION Input
On entry: the tolerance, tol, of the convergence criteria (3.6) and (3.7); it should be an estimate of the largest relative error in the elements of $A$. For example, if the
elements of $A$ are correct to about 4 significant figures, then $\text{ATOL}$ should be set to about $5 \times 10^{-4}$. If $\text{ATOL}$ is supplied as less than $(\epsilon)$, where $(\epsilon)$ is the machine precision, then the value $(\epsilon)$ is used in place of $\text{ATOL}$.

9: \textbf{BTOL -- DOUBLE PRECISION} \hspace{1cm} \text{Input}

On entry: the tolerance, $\text{tol}$, of the convergence criterion (3.6); it should be an estimate of the largest relative error in the elements of $B$. For example, if the elements of $B$ are correct to about 4 significant figures, then $\text{BTOL}$ should be set to about $5 \times 10^{-4}$. If $\text{BTOL}$ is supplied as less than $(\epsilon)$, where $(\epsilon)$ is the machine precision, then the value $(\epsilon)$ is used in place of $\text{BTOL}$.

10: \textbf{CONLIM -- DOUBLE PRECISION} \hspace{1cm} \text{Input}

On entry: the value $\lim$ of equation (3.8); it should be an upper limit on the condition number of $A$. $\text{CONLIM}$ should not normally be chosen much larger than $1.0/\text{ATOL}$. If $\text{CONLIM}$ is supplied as zero then the value $1.0/(\epsilon)$, where $(\epsilon)$ is the machine precision, is used in place of $\text{CONLIM}$.

11: \textbf{ITNLIM -- INTEGER} \hspace{1cm} \text{Input}

On entry: an upper limit on the number of iterations. If $\text{ITNLIM} \leq 0$, then the value $N$ is used in place of $\text{ITNLIM}$, but for ill-conditioned problems a higher value of $\text{ITNLIM}$ is likely to be necessary.

12: \textbf{MSGLVL -- INTEGER} \hspace{1cm} \text{Input}

On entry: the level of printing from F04QAF. If $\text{MSGLVL} \leq 0$, then no printing occurs, but otherwise messages will be output on the advisory message channel (see X04ABF). A description of the printed output is given in Section 5.2 below. The level of printing is determined as follows:

\begin{itemize}
  \item[$\text{MSGLVL} \leq 0$] No printing.
  \item[$\text{MSGLVL} = 1$] A brief summary is printed just prior to return from F04QAF.
  \item[$\text{MSGLVL} \geq 2$] A summary line is printed periodically to monitor the
progress of F04QAF, together with a brief summary just prior to return from F04QAF.

13: ITN -- INTEGER Output
On exit: the number of iterations performed.

14: ANORM -- DOUBLE PRECISION Output
On exit: an estimate of \( ||A|| \) for the matrix \( A \) of equation (3.4).

15: ACOND -- DOUBLE PRECISION Output
On exit: an estimate of \( \text{cond}(A) \) which is a lower bound.

16: RNORM -- DOUBLE PRECISION Output
On exit: an estimate of \( ||r|| \) for the residual, \( r \), of equation (3.5) corresponding to the solution \( x \) returned in \( X \). Note that \( ||r|| \) is the function being minimized.

17: ARNORM -- DOUBLE PRECISION Output
\( T \)
On exit: an estimate of the \( ||A^T r|| \) corresponding to the solution \( x \) returned in \( X \).

18: XNORM -- DOUBLE PRECISION Output
On exit: an estimate of \( ||x|| \) for the solution \( x \) returned in \( X \).

19: WORK(2*N) -- DOUBLE PRECISION array Workspace

20: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace
RWORK is not used by F04QAF, but is passed directly to routine APROD and may be used to pass information to that routine.

21: LRWORK -- INTEGER Input
On entry: the length of the array RWORK as declared in the (sub)program from which F04QAF is called. Constraint: \( LRWORK \geq 1 \).

22: IWORK(LIWORK) -- INTEGER array User Workspace
IWORK is not used by F04QAF, but is passed directly to routine APROD and may be used to pass information to that
routine.

23: LIWORK -- INTEGER
Input
On entry: the length of the array IWORK as declared in the
(sub)program from which F04QAF is called. Constraint: LIWORK
>= 1.

24: INFORM -- INTEGER
Output
On exit: the reason for termination of F04QAF as follows:

INFORM = 0
The exact solution is x=0. No iterations are performed
in this case.

INFORM = 1
The termination criterion of equation (3.6) has been
satisfied with tol and tol as the values supplied in
1 2
ATOL and BTOL respectively.

INFORM = 2
The termination criterion of equation (3.7) has been
satisfied with tol as the value supplied in ATOL.
1

INFORM = 3
The termination criterion of equation (3.6) has been
satisfied with tol and/or tol as the value (epsilon)
1 2
, where (epsilon) is the machine precision. One or
both of the values supplied in ATOL and BTOL must have
been less than (epsilon) and was too small for this
machine.

INFORM = 4
The termination criterion of equation (3.7) has been
satisfied with tol as the value (epsilon), where
1
(epsilon) is the machine precision. The value supplied
in ATOL must have been less than (epsilon) and was too
small for this machine.

The values INFORM = 5, INFORM = 6 and INFORM = 7 correspond
to failure with IFAIL = 2, IFAIL = 3 and IFAIL = 4
respectively (see Section 6) and when IFAIL is negative
INFORM will be set to the same negative value.

25: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
5.1. Description of the printed output

When MSGLVL > 0, then F04QAF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF).

When MSGLVL >= 2 then a summary line is printed periodically giving the following information:

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITN</td>
<td>Iteration number, ( k ).</td>
</tr>
<tr>
<td>X(1)</td>
<td>The first element of the current iterate ( x_k ).</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>The current value of the function, ( (\rho) ), being minimized.</td>
</tr>
<tr>
<td>COMPAT</td>
<td>An estimate of (</td>
</tr>
<tr>
<td>INCOMPAT</td>
<td>An estimate of (</td>
</tr>
<tr>
<td>NRM(ABAR)</td>
<td>A monotonically increasing estimate of (</td>
</tr>
<tr>
<td>COND(ABAR)</td>
<td>A monotonically increasing estimate of the condition number ( \text{cond}(A) ).</td>
</tr>
</tbody>
</table>

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL< 0
A negative value of IFAIL indicates an exit from F04QAF because the user has set MODE negative in APROD. The value of IFAIL will be the same as the user’s setting of MODE.

IFAIL= 1
On entry M < 1,
or N < 1,
or LRWORK < 1,
or LIWORK < 1.

IFAIL= 2
The condition of equation (3.8) has been satisfied for the value of c supplied in CONLIM. If this failure is unexpected the user should check that APROD is working correctly. Although conditions (3.6) or (3.7) have not been satisfied, the values returned in RNORM, ARNORM and XNORM may nevertheless indicate that an acceptable solution has been reached.

IFAIL= 3
The conditions of equation (3.8) has been satisfied for the value c = 1.0/(epsilon), where (epsilon) is the machine precision. The matrix A is nearly singular or rank deficient and the problem is too ill-conditioned for this machine. If this failure is unexpected, the user should check that APROD is working correctly.

IFAIL= 4
The limit on the number of iterations has been reached. The number of iterations required by F04QAF and the condition of the matrix A can depend strongly on the scaling of the problem. Poor scaling of the rows and columns of A should be avoided whenever possible.

7. Accuracy

When the problem is compatible, the computed solution x will satisfy the equation
r=b-Ax,

where an estimate of \( ||r|| \) is returned in the parameter RNORM. When the problem is incompatible, the computed solution \( x \) will satisfy the equation

\[
T \quad A r = e,
\]

where an estimate of \( ||e|| \) is returned in the parameter ARNORM. See also Section 6.2 of Paige and Saunders [1].

8. Further Comments

The time taken by the routine is likely to be principally determined by the time taken in APROD, which is called twice on each iteration, once with MODE = 1 and once with MODE = 2. The time taken per iteration by the remaining operations in F04QAF is approximately proportional to \( \max(m,n) \).

The Lanczos process will usually converge more quickly if \( A \) is pre-conditioned by a non-singular matrix \( M \) that approximates \( A \) in some sense and is also chosen so that equations of the form \( My = c \) can efficiently be solved for \( y \). Some discussion of pre-conditioning in the context of symmetric matrices is given in Section 3 of the document for F04MBF. In the context of F04QAF, problem (3.1) is equivalent to

\[
-1 (AM)^{-1} y = b, \quad M x = y
\]

and problem (3.2) is equivalent to minimizing

\[
-1 \quad (\rho) = ||r||, \quad r = b - (AM)^{-1} y, \quad M x = y.
\]

Note that the normal matrix \( (AM) (AM)^{-1} = M (A A)^{-1} M \) so that the pre-conditioning \( AM \) is equivalent to the pre-conditioning \( M (A A)^{-1} \) of the normal matrix \( A A \).

Pre-conditioning can be incorporated into F04QAF simply by coding

\[
-1 -T T -1 -T T -1
\]

the routine APROD to compute \( y + AM x \) and \( x + M A y \) in place of

\[
y + Ax \quad \text{and} \quad x + A y
\]

respectively, and then solving the equations \( M x = y \).
for \( x \) on return from F04QAF, \( y + \mathbf{AM} \ x \) should be computed by solving \( \mathbf{Mz} = \mathbf{x} \) for \( z \) and then computing \( y + \mathbf{Az} \), and \( \mathbf{x} + \mathbf{M} \ y \) should be computed by solving \( \mathbf{Mz} = \mathbf{A} \ y \) for \( z \) and then forming \( \mathbf{x} + \mathbf{z} \).

9. Example

To solve the linear least-squares problem

\[
\minimize \ (\rho) = ||\mathbf{r}||, \quad \mathbf{r} = \mathbf{b} - \mathbf{A} \mathbf{x}
\]

where \( \mathbf{A} \) is the 13 by 12 matrix and \( \mathbf{b} \) is the 13 element vector given by

\[
\mathbf{A} = \begin{pmatrix}
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[
\mathbf{A} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
\end{pmatrix}
\]

\[
\mathbf{b} = -h \ (0) \\
(1) \\
(0) \\
(-3) \\
(-h)
\]

with \( h = 0.1 \).

Such a problem can arise by considering the Neumann problem on a rectangle

\[
\delta u
\]
\[ \begin{align*}
\text{(delta)}u & \quad 2 \quad \text{(delta)}u \\
\text{(delta)}n & \quad \text{=} 0 \quad \text{(nabla)} u = g(x,y) \quad \text{(delta)}n \\
\end{align*} \]

where \( C \) is the boundary of the rectangle, and discretising as illustrated below with the square mesh.

Please see figure in printed Reference Manual

The 12 by 12 symmetric part of \( A \) represents the difference equations and the final row comes from the normalising condition. The example program has \( g(x,y)=1 \) at all the internal mesh points, but apart from this is written in a general manner so that the number of rows (NROWS) and columns (NCOLS) in the grid can readily be altered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

NagLinearEquationSolvingPackage (NAGF04)
Exports:
f04adf  f04arf  f04asf  f04atf  f04axf
f04af  f04jgf  f04mcf  f04qaf

--- package NAGF04 NagLinearEquationSolvingPackage ---

)abbrev package NAGF04 NagLinearEquationSolvingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:31 1994
++ Description:
++ This package uses the NAG Library to solve the matrix equation \( AX=B \),
++ where \( B \) may be a single vector or a matrix of multiple right-hand sides.
++ The matrix \( A \) may be real, complex, symmetric, Hermitian positive-
++ definite, or sparse. It may also be rectangular, in which case a
++ least-squares solution is obtained.

NagLinearEquationSolvingPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==>
with
  f04adf : (Integer,Matrix Complex DoubleFloat,Integer,Integer,_,Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
  ++ f04adf(ia,b,ib,n,m,ic,a,ifail)
  ++ calculates the approximate solution of a set of complex
  ++ linear equations with multiple right-hand sides, using an LU
  ++ factorization with partial pivoting.
  ++ See \downlink{Manual Page}{manpageXXf04adf}.
  f04arf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer) -> Result
  ++ f04arf(ia,b,n,a,ifail)
  ++ calculates the approximate solution of a set of real
  ++ linear equations with a single right-hand side, using an LU
  ++ factorization with partial pivoting.
  ++ See \downlink{Manual Page}{manpageXXf04arf}.
  f04asf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer) -> Result
  ++ f04asf(ia,b,n,a,ifail)
  ++ calculates the accurate solution of a set of real
  ++ symmetric positive-definite linear equations with a single right-
  ++ hand side, \( Ax=b \), using a Cholesky factorization and iterative
  ++ refinement.
  ++ See \downlink{Manual Page}{manpageXXf04asf}.
  f04atf : (Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,Integer) -> Result
  ++ f04atf(a,ia,b,n,iaa,ifail)
  ++ calculates the accurate solution of a set of real linear
++ equations with a single right-hand side, using an LU
++ factorization with partial pivoting, and iterative refinement.
++ See \downlink{Manual Page}{manpageXXf04atf}.
f04axf : (Integer,Matrix DoubleFloat,Integer,Matrix Integer,_
       Matrix Integer,Matrix DoubleFloat) -> Result
++ f04axf(n,a,licn,icn,ikeep,mtype,idisp,rhs)
++ calculates the approximate solution of a set of real
++ sparse linear equations with a single right-hand side, Ax=b or
++ T
++ A x=b, where A has been factorized by F01BRF or F01BSF.
++ See \downlink{Manual Page}{manpageXXf04axf}.
f04faf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
       Matrix DoubleFloat,Integer) -> Result
++ f04faf(job,n,d,e,b,ifail)
++ calculates the approximate solution of a set of real
++ symmetric positive-definite tridiagonal linear equations.
++ See \downlink{Manual Page}{manpageXXf04faf}.
f04jgf : (Integer,Integer,Integer,DoubleFloat,_
       Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f04jgf(m,n,nra,tol,a,b,ifail)
++ finds the solution of a linear least-squares problem, Ax=b
++ , where A is a real m by n (m>\geq n) matrix and b is an n element
++ vector. If the matrix of observations is not of full rank, then
++ the minimal least-squares solution is returned.
++ See \downlink{Manual Page}{manpageXXf04jgf}.
f04maf : (Integer,Integer,Matrix DoubleFloat,Integer,_
       Matrix Integer,Integer,Matrix Integer,Matrix DoubleFloat,_
       Matrix Integer,Integer,Integer,Integer) -> Result
++ f04maf(n,nz,avals,licn,irn,lirn,icn,wkeep,ikeep,
++ inform,b,acc,noits,ifail)
++ e a sparse symmetric positive-definite system of linear
++ equations, Ax=b, using a pre-conditioned conjugate gradient
++ method, where A has been factorized by F01MAF.
++ See \downlink{Manual Page}{manpageXXf04maf}.
f04mbf : (Integer,Matrix DoubleFloat,Boolean,DoubleFloat,_
       Integer,Integer,Integer,Integer,DoubleFloat,Integer,_
       Union(fn:FileName,fp:Asp28(APROD)),_
       Union(fn:FileName,fp:Asp34(MSOLVE))) -> Result
++ f04mbf(n,b,precon,shift,itnlim,msglvl,lrwork,_
++ livwork,rtol,ifail,aprod,msolve)
++ solves a system of real sparse symmetric linear equations
++ using a Lanczos algorithm.
++ See \downlink{Manual Page}{manpageXXf04mbf}.
f04mcf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
       Matrix Integer,Integer,Matrix DoubleFloat,Integer,Integer,_
       Integer,Integer) -> Result
++ f04mcf(n,al,lal,d,nrow,ir,b,nrblt,iselct,nrx,ifail)
++ computes the approximate solution of a system of real
++ linear equations with multiple right-hand sides, AX=B, where A
++ is a symmetric positive-definite variable-bandwidth matrix, which
++ has previously been factorized by F01MCF. Related systems may
++ also be solved.
++ See \downlink{Manual Page}{manpageXXf04mcf}.
++ See \downlink{Manual Page}{manpageXXf04qaf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)

f04adf(iaArg:Integer,bArg:Matrix Complex DoubleFloat,ibArg:Integer,_
          nArg:Integer,mArg:Integer,icArg:Integer,_
          aArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result ==
          [(invokeNagman(NIL$Lisp,_
                         "f04adf",_
                         ["ia"::S,"ib"::S,"n"::S,"m"::S,"ic"::S _
                          ["c"::S,"wkspace"::S]$Lisp,_
                          ["double"::S,["wkspace"::S,"n"::S]$Lisp]$Lisp_ 
                          ,["integer"::S,"ia"::S,"ib"::S,"n"::S,"m"::S _
                          ,"ic"::S,"ifail"::S]$Lisp_ 
                          ,["double complex"::S,"b"::S,"ib"::S,"m"::S]$Lisp_ ,
                          "$Lisp_ _
                          ["c"::S,"a"::S,"ifail"::S]$Lisp_ _
                          [(iaArg::Any,ibArg::Any,nArg::Any,mArg::Any,icArg::Any,_
                          ifailArg::Any,bArg::Any,aArg::Any )]_
\[
\begin{align*}
\text{f04arf}(\text{iArg: Integer, bArg: Matrix DoubleFloat, nArg: Integer,} \\
\text{aArg: Matrix DoubleFloat, ifailArg: Integer}): \text{ Result } &= \text{ invokeNagman NIL Lisp,} \\
\text{"f04arf",} \\
["\text{ia": S, "n": S, "ifail": S, "b": S, "c": S, "a": S, "wkspace": S}]$\text{Lisp,} \\
["\text{c": S, "wkspace": S}]$\text{Lisp,} \\
["\text{double": S, ["b": S, "n": S}]$\text{Lisp, ["c": S, "n": S}]$\text{Lisp,} \\
["a": S, "ia": S, "n": S}]$\text{Lisp, ["wkspace": S, "n": S}]$\text{Lisp}]$\text{Lisp,} \\
["\text{integer": S, "ia": S, "n": S, "ifail": S}]$\text{Lisp} \\
]\text{Lisp,} \\
["\text{c": S, "a": S, "ifail": S}]$\text{Lisp,} \\
[(\text{iArg:: Any, nArg:: Any, ifailArg:: Any, bArg:: Any, aArg:: Any }]) \\
@\text{List Any)}$\text{Lisp)$\text{Lisp})_}
\end{align*}
\]

\[
\begin{align*}
\text{f04asf}(\text{iArg: Integer, bArg: Matrix DoubleFloat, nArg: Integer,} \\
\text{aArg: Matrix DoubleFloat, ifailArg: Integer}): \text{ Result } &= \text{ invokeNagman NIL Lisp,} \\
\text{"f04asf",} \\
["\text{ia": S, "n": S, "ifail": S, "b": S, "c": S, "a": S, "wk1": S, "wk2": S}]$\text{Lisp,} \\
["\text{c": S, "wk1": S, "wk2": S}]$\text{Lisp,} \\
["\text{double": S, ["b": S, "n": S}]$\text{Lisp, ["c": S, "n": S}]$\text{Lisp,} \\
["a": S, "ia": S, "n": S}]$\text{Lisp, ["wk1": S, "n": S}]$\text{Lisp,} \\
["wk2": S, "n": S}]$\text{Lisp}]$\text{Lisp,} \\
["\text{integer": S, "ia": S, "n": S, "ifail": S}]$\text{Lisp} \\
]\text{Lisp,} \\
["\text{c": S, "a": S, "ifail": S}]$\text{Lisp,} \\
[(\text{iArg:: Any, nArg:: Any, ifailArg:: Any, bArg:: Any, aArg:: Any }]) \\
@\text{List Any)}$\text{Lisp)$\text{Lisp})_}
\end{align*}
\]

\[
\begin{align*}
\text{f04atf}(\text{aArg: Matrix DoubleFloat, iArg: Integer, bArg: Matrix DoubleFloat,} \\
nArg: Integer, iaaArg: Integer, ifailArg: Integer): \text{ Result } &= \text{ invokeNagman NIL Lisp,} \\
\text{"f04atf",} \\
["\text{ia": S, "n": S, "ia": S, "ifail": S, "a": S, "b": S, "c": S,} \\
\text{"aa": S, "wks1": S, "wks2": S}]$\text{Lisp,} \\
["\text{c": S, "wks1": S, "wks2": S}]$\text{Lisp,} \\
["\text{double": S, ["a": S, "ia": S, "n": S}]$\text{Lisp,} \\
["b": S, "n": S}]$\text{Lisp, ["c": S, "n": S}]$\text{Lisp,} \\
["aa": S, "iaa": S, "n": S}]$\text{Lisp, ["wks1": S, "n": S}]$\text{Lisp,} \\
["wks2": S, "n": S}]$\text{Lisp}]$\text{Lisp,} \\
["\text{integer": S, "ia": S, "n": S, "iaa": S, "ifail": S}]$\text{Lisp} \\
["\text{c": S, "aa": S, "ifail": S}]$\text{Lisp,} \\
[(\text{iArg:: Any, nArg:: Any, iaaArg:: Any, ifailArg:: Any,} \\
aArg:: Any, bArg:: Any)]@\text{List Any)}$\text{Lisp)$\text{Lisp})_}
\end{align*}
\]
pretend List (Record(key:Symbol,entry:Any))$Result

f04axf(nArg:Integer,aArg:Matrix DoubleFloat,licnArg:Integer,_
icnArg:Matrix Integer,ikeepArg:Matrix Integer,mtypeArg:Integer,_
idispArg:Matrix Integer,rhsArg:Matrix DoubleFloat): Result ==
[(invokeNagman(NIL$Lisp,_
"f04axf",_
["n":"S","licn":"S","mtype":"S","resid":"S","a":"S","icn":"S_,
"ikeep":"S","idisp":"S","rhs":"S","w":"S"]$Lisp,_
["resid":"S","w":"S"]$Lisp,_
[["double":"S","a":"S","licn":"S"]$Lisp,"resid":"S_,
,["rhs":"S","n":"S"]$Lisp,["w":"S","n":"S"]$Lisp]$Lisp,_
,[["integer":"S","n":"S","licn":"S","icn":"S"]$Lisp_,
,[["ikeep":"S","n":"S",5$Lisp]$Lisp,$Lisp,_
]$Lisp,_
["resid":"S","rhs":"S"]$Lisp,_
[[nArg::Any,licnArg::Any,mtypeArg::Any,aArg::Any,icnArg::Any_,
ikeepArg::Any,idispArg::Any,rhsArg::Any ]]_
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04faf(jobArg:Integer,nArg:Integer,dArg:Matrix DoubleFloat,_
eArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f04faf",_
["job":"S","n":"S","ifail":"S","d":"S","e":"S","b":"S"]$Lisp,_
[])$Lisp,_
[["double":"S","d":"S","n":"S"]$Lisp,["e":"S","n":"S"]$Lisp_,
,[["b":"S","n":"S"]$Lisp]$Lisp]$Lisp,_
,[["integer":"S","job":"S","n":"S","ifail":"S"]$Lisp_]
]$Lisp,_
["d":"S","e":"S","b":"S","ifail":"S"]$Lisp,_
[[jobArg::Any,nArg::Any,ifailArg::Any,ifailArg::Any,jobArg::Any,bArg::Any]]_
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04jgf(mArg:Integer,nArg:Integer,nraArg:Integer,_
tolArg:DoubleFloat,lworkArg:Integer,aArg:Matrix DoubleFloat,_
bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f04jgf",_
["m":"S","n":"S","nra":"S","tol":"S","lwork":"S_,
,"svd":"S","sigma":"S","irank":"S","ifail":"S","work":"S_,
"a":"S","b":"S"]$Lisp,_
["svd":"S","sigma":"S","irank":"S","work":"S"]$Lisp,_
[["double":"S","tol":"S","sigma":"S","work":"S"]$Lisp_,
,[["a":"S","nra":"S","n":"S"]$Lisp,["b":"S","m":"S"]$Lisp]$Lisp,_
,[["integer":"S","m":"S","n":"S","nra":"S","lwork":"S_


-- if both asps are AXIOM generated we do not need lrwork liwork
-- and will set to 1.
-- else believe the user but check that they are >0.
if (aprodArg case fp) and (msolveArg case fp)
then
    lrworkArg:=1
    liworkArg:=1
else
    lrworkArg:=max(1,lrworkArg)
    liworkArg:=max(1,liworkArg)
pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
if aprodArg case fn
    then outputAsFortran(aprodArg.fn)
else outputAsFortran(aprodArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(msolveFilename := aspFilename "msolve")$FOP
if msolveArg case fn
    then outputAsFortran(msolveArg.fn)
else outputAsFortran(msolveArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([aprodFilename,msolveFilename]$Lisp,_,
"f04mcf",_
[(nArg::Any,preconArg::Any,shiftArg::Any,itnlimArg::Any_ ,msglvlArg::Any,lrworkArg::Any,irworkArg::Any,rtolArg::Any_ ,ifailArg::Any,bArg::Any )]_ @List Any]$Lisp)$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

f04mcf(nArg:Integer,alArg:Matrix DoubleFloat,lalArg:Integer,_
dArg:Matrix DoubleFloat,nrowArg:Matrix Integer,irArg:Integer,_
bArg:Matrix DoubleFloat,nrbArg:Integer,iselctArg:Integer,_
nrxAArg:Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp_ ,
"f04mcf",_ 
["x":S]$Lisp_ ,
CHAPTER 15. CHAPTER N

["x":S,"ifail":S]$Lisp_,
bArg":Any ]\@List Any]$Lisp)_pretend List (Record(key:Symbol,entry:Any))$Result

f04qaf(mArg:Integer,nArg:Integer,dampArg:DoubleFloat,_
atolArg:DoubleFloat,btolArg:DoubleFloat,conlimArg:DoubleFloat,_
itnlimArg:Integer,msglvlArg:Integer,_
lworkArg:Integer,bArg:Matrix DoubleFloat,ifailArg:Integer,_
aprodArg:Union(fn:FileName,fp:Asp30(APROD))): Result ==
pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
if aprodArg case fn
  then outputAsFortran(aprodArg.fn)
  else outputAsFortran(aprodArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([aprodFilename]$Lisp,_
"f04qaf",_
["m":S,"n":S,"damp":S,"atol":S,"btol":S_,
,"rwork"::S,"rwork"::S_,
,"iwork"::S]$Lisp_,$Lisp_,
,"iwork":S,"aprod":S]$Lisp_,$Lisp_,
"anorm":S,"acond":S,"rnorm":S,"arnorm":S,"xnorm":S,
,"b":S,"m":S]$Lisp_,$Lisp_,
,"aprod":S]$Lisp_,$Lisp_,
["integer":S,"m":S,"n":S,"itnlim":S,"msglvl":S_,
["iwork":S,"lwork":S]$Lisp]$Lisp_,$Lisp_,$Lisp_,
"bArg":Any ]@List Any]$Lisp)_pretend List (Record(key:Symbol,entry:Any))$Result


package NAGSP NAGLinkSupportPackage

-- NAGLinkSupportPackage.input --

)set break resume
)sys rm -f NAGLinkSupportPackage.output
)spool NAGLinkSupportPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NAGLinkSupportPackage
--E 1

)spool
)lisp (bye)

---

-- NAGLinkSupportPackage.help --

====================================================================
NAGLinkSupportPackage examples
====================================================================

Support functions for the NAG Library Link functions

See Also:
  o )show NAGLinkSupportPackage

---
NAGLinkSupportPackage (NAGSP)

Exports:

aspFilename  checkPrecision  dimensionsOf  fortranCompilerName
fortranLinkerArgs  restorePrecision

package NAGSP NAGLinkSupportPackage

)abbrev package NAGSP NAGLinkSupportPackage
++ Author: Mike Devar and Godfrey Nolan
++ Date Created: March 1993
++ Date Last Updated: October 6 1994
++ Description:
++ Support functions for the NAG Library Link functions

NAGLinkSupportPackage() : exports == implementation where

exports ==> with
  fortranCompilerName : () -> String
    ++ fortranCompilerName() returns the name of the currently selected
    ++ Fortran compiler
  fortranLinkerArgs  : () -> String
    ++ fortranLinkerArgs() returns the current linker arguments
  aspFilename : String -> String
    ++ aspFilename("f") returns a String consisting of "f" suffixed with
    ++ an extension identifying the current AXIOM session.
  dimensionsOf  : (Symbol, Matrix DoubleFloat) -> SExpression
    ++ dimensionsOf(s,m) \undocumented{}
  dimensionsOf  : (Symbol, Matrix Integer) -> SExpression
    ++ dimensionsOf(s,m) \undocumented{}
  checkPrecision : () -> Boolean
    ++ checkPrecision() \undocumented{}
  restorePrecision : () -> Void
    ++ restorePrecision() \undocumented{}

implementation ==> add

makeAs: (Symbol,Symbol) -> Symbol
changeVariables: (Expression Integer, Symbol) -> Expression Integer
changeVariablesF: (Expression Float, Symbol) -> Expression Float

import String
import Symbol

checkPrecision(): Boolean ==
  (_.$fortranPrecision$Lisp = "single":Symbol) and _
  (_.$nagEnforceDouble$Lisp) =>
    systemCommand("set fortran precision double")$MoreSystemCommands
    if _.$nagMessages$Lisp then
      print("*** Warning: Resetting fortran precision to double")_
      $PrintPackage
    true
  false

restorePrecision(): Void ==
  systemCommand("set fortran precision single")$MoreSystemCommands
  if _.$nagMessages$Lisp then
    print("** Warning: Restoring fortran precision to single")$PrintPackage
  void()

uniqueId : String := ""
counter : Integer := 0
getUniqueId(): String ==
  if uniqueId = "" then
    uniqueId := concat(getEnv("HOST")$Lisp,getEnv("SPADNUM")$Lisp)
    concat(uniqueId,string (counter:=counter+1))

fortranCompilerName() == string $_.$fortranCompilerName$Lisp
fortranLinkerArgs() == string $_.$fortranLibraries$Lisp

aspFilename(f: String): String == concat ["/tmp/", f, getUniqueId(), ".f"]

dimensionsOf(u: Symbol, m: Matrix DoubleFloat): SExpression ==
  [u, nrows m, ncols m]$Lisp
dimensionsOf(u: Symbol, m: Matrix Integer): SExpression ==
  [u, nrows m, ncols m]$Lisp

—— NAGSP.dotabb ——

"NAGSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGSP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGSP" -> "ALIST"
package NAGD01 NagIntegrationPackage

— NagIntegrationPackage.input —

)set break resume
)sys rm -f NagIntegrationPackage.output
)spool NagIntegrationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagIntegrationPackage
--E 1

)spool
)lisp (bye)

— NagIntegrationPackage.help —

This package uses the NAG Library to calculate the numerical value of
definite integrals in one or more dimensions and to evaluate
weights and abscissae of integration rules.

D01(3NAG) Foundation Library (12/10/92) D01(3NAG)
D01 -- Quadrature
Introduction -- D01
Chapter D01
Quadrature

1. Scope of the Chapter

This chapter provides routines for the numerical evaluation of
definite integrals in one or more dimensions and for evaluating
weights and abscissae of integration rules.

2. Background to the Problems

The routines in this chapter are designed to estimate:

(a) the value of a one-dimensional definite integral of the form:

\[ \int_{a}^{b} f(x) \, dx \]  

(1)
where $f(x)$ is defined by the user, either at a set of points $(x_i, f(x_i))$, for $i=1,2,...,n$ where $a=x_1 < x_2 < ... < x_n = b$, or in the form of a function; and the limits of integration $a,b$ may be finite or infinite.

Some methods are specially designed for integrands of the form

$$f(x) = w(x)g(x)$$

which contain a factor $w(x)$, called the weight-function, of a specific form. These methods take full account of any peculiar behaviour attributable to the $w(x)$ factor.

(b) the value of a multi-dimensional definite integral of the form:

$$\int_{R} \int_{a}^{b} \cdots \int_{x_1}^{x_n} f(x_1, x_2, \ldots, x_n) \, dx_1 \cdots dx_n$$

where $f(x_1, x_2, \ldots, x_n)$ is a function defined by the user and $R$ is some region of $n$-dimensional space.

The simplest form of $R$ is the $n$-rectangle defined by:

$$a_i \leq x_i \leq b_i, \quad i=1,2,...,n$$

where $a_i$ and $b_i$ are constants. When $a_i$ and $b_i$ are functions of $x_j$ ($j<i$), the region can easily be transformed to the rectangular form (see Davis and Rabinowitz [1] page 266).

Some of the methods described incorporate the transformation procedure.

2.1. One-dimensional Integrals

To estimate the value of a one-dimensional integral, a quadrature rule uses an approximation in the form of a weighted sum of integrand values, i.e.,

$$\int_{a}^{b} f(x) \, dx \approx \sum_{i=1}^{N} w_i f(x_i).$$

where $f(x)$ is defined by the user, either at a set of points $(x_i, f(x_i))$, for $i=1,2,...,n$ where $a=x_1 < x_2 < ... < x_n = b$, or in the form of a function; and the limits of integration $a,b$ may be finite or infinite.

Some methods are specially designed for integrands of the form

$$f(x) = w(x)g(x)$$

which contain a factor $w(x)$, called the weight-function, of a specific form. These methods take full account of any peculiar behaviour attributable to the $w(x)$ factor.

(b) the value of a multi-dimensional definite integral of the form:

$$\int_{R} \int_{a}^{b} \cdots \int_{x_1}^{x_n} f(x_1, x_2, \ldots, x_n) \, dx_1 \cdots dx_n$$

where $f(x_1, x_2, \ldots, x_n)$ is a function defined by the user and $R$ is some region of $n$-dimensional space.

The simplest form of $R$ is the $n$-rectangle defined by:

$$a_i \leq x_i \leq b_i, \quad i=1,2,...,n$$

where $a_i$ and $b_i$ are constants. When $a_i$ and $b_i$ are functions of $x_j$ ($j<i$), the region can easily be transformed to the rectangular form (see Davis and Rabinowitz [1] page 266).

Some of the methods described incorporate the transformation procedure.

2.1. One-dimensional Integrals

To estimate the value of a one-dimensional integral, a quadrature rule uses an approximation in the form of a weighted sum of integrand values, i.e.,

$$\int_{a}^{b} f(x) \, dx \approx \sum_{i=1}^{N} w_i f(x_i).$$
The points $x_i$ within the interval $[a,b]$ are known as the abscissae, and the $w_i$ are known as the weights.

More generally, if the integrand has the form (2), the corresponding formula is

$$\int_{a}^{b} w(x)g(x)dx = \sum_{i=1}^{N} w_i g(x_i). \quad (6)$$

If the integrand is known only at a fixed set of points, these points must be used as the abscissae, and the weighted sum is calculated using finite-difference methods. However, if the functional form of the integrand is known, so that its value at any abscissa is easily obtained, then a wide variety of quadrature rules are available, each characterised by its choice of abscissae and the corresponding weights.

The appropriate rule to use will depend on the interval $[a,b]$ – whether finite or otherwise – and on the form of any $w(x)$ factor in the integrand. A suitable value of $N$ depends on the general behaviour of $f(x)$; or of $g(x)$, if there is a $w(x)$ factor present.

Among possible rules, we mention particularly the Gaussian formulae, which employ a distribution of abscissae which is optimal for $f(x)$ or $g(x)$ of polynomial form.

The choice of basic rules constitutes one of the principles on which methods for one-dimensional integrals may be classified. The other major basis of classification is the implementation strategy, of which some types are now presented.

(a) Single rule evaluation procedures

A fixed number of abscissae, $N$, is used. This number and the particular rule chosen uniquely determine the weights and abscissae. No estimate is made of the accuracy of the result.

(b) Automatic procedures

The number of abscissae, $N$, within $[a,b]$ is gradually increased until consistency is achieved to within a level of accuracy (absolute or relative) requested by the user. There are essentially two ways of doing this; hybrid forms
of these two methods are also possible:

(i) whole interval procedures (non-adaptive)

A series of rules using increasing values of N are successively applied over the whole interval \([a,b]\). It is clearly more economical if abscissae already used for a lower value of N can be used again as part of a higher-order formula. This principle is known as optimal extension. There is no overlap between the abscissae used in Gaussian formulae of different orders. However, the Kronrod formulae are designed to give an optimal \((2N+1)\)-point formula by adding \((N+1)\) points to an \(N\)-point Gauss formula. Further extensions have been developed by Patterson.

(ii) adaptive procedures

The interval \([a,b]\) is repeatedly divided into a number of sub-intervals, and integration rules are applied separately to each sub-interval. Typically, the subdivision process will be carried further in the neighbourhood of a sharp peak in the integrand, than where the curve is smooth. Thus, the distribution of abscissae is adapted to the shape of the integrand.

Subdivision raises the problem of what constitutes an acceptable accuracy in each sub-interval. The usual global acceptability criterion demands that the sum of the absolute values of the error estimates in the sub-intervals should meet the conditions required of the error over the whole interval. Automatic extrapolation over several levels of subdivision may eliminate the effects of some types of singularities.

An ideal general-purpose method would be an automatic method which could be used for a wide variety of integrands, was efficient (i.e., required the use of as few abscissae as possible), and was reliable (i.e., always gave results within the requested accuracy). Complete reliability is unobtainable, and generally higher reliability is obtained at the expense of efficiency, and vice versa. It must therefore be emphasised that the automatic routines in this chapter cannot be assumed to be 100% reliable. In general, however, the reliability is very high.

2.2. Multi-dimensional Integrals

A distinction must be made between cases of moderately low dimensionality (say, up to 4 or 5 dimensions), and those of higher dimensionality. Where the number of dimensions is limited,
a one-dimensional method may be applied to each dimension, according to some suitable strategy, and high accuracy may be obtainable (using product rules). However, the number of integrand evaluations rises very rapidly with the number of dimensions, so that the accuracy obtainable with an acceptable amount of computational labour is limited; for example a product of 3-point rules in 20 dimensions would require more than 10 integrand evaluations. Special techniques such as the Monte Carlo methods can be used to deal with high dimensions.

(a) Products of one-dimensional rules

Using a two-dimensional integral as an example, we have

\[
\begin{align*}
\int_{a}^{b} \int_{a}^{b} f(x,y) \, dy \, dx &= \sum_{i=1}^{N} \sum_{j=1}^{N} w_{i} v_{i,j} f(x_{i},y_{j}) \\
\end{align*}
\]

\[
\int_{a}^{b} \int_{a}^{b} f(x,y) \, dy \, dx &= \sum_{i=1}^{N} \sum_{j=1}^{N} w_{i} v_{i,j} f(x_{i},y_{j}) \\
\]

where \((w_{i},x_{i})\) and \((v_{i,j},y_{j})\) are the weights and abscissae of the rules used in the respective dimensions.

A different one-dimensional rule may be used for each dimension, as appropriate to the range and any weight function present, and a different strategy may be used, as appropriate to the integrand behaviour as a function of each independent variable.

For a rule-evaluation strategy in all dimensions, the formula (8) is applied in a straightforward manner. For automatic strategies (i.e., attempting to attain a requested accuracy), there is a problem in deciding what accuracy must be requested in the inner integral(s). Reference to formula (7) shows that the presence of a limited but random error in the y-integration for different values of \(x\) can produce a 'jagged' function of \(x\), which may be difficult to integrate to the desired accuracy and for this reason products of automatic one-dimensional
routines should be used with caution (see also Lyness [3]).

(b) Monte Carlo methods

These are based on estimating the mean value of the integrand sampled at points chosen from an appropriate statistical distribution function. Usually a variance reducing procedure is incorporated to combat the fundamentally slow rate of convergence of the rudimentary form of the technique. These methods can be effective by comparison with alternative methods when the integrand contains singularities or is erratic in some way, but they are of quite limited accuracy.

(c) Number theoretic methods

These are based on the work of Korobov and Conroy and operate by exploiting implicitly the properties of the Fourier expansion of the integrand. Special rules, constructed from so-called optimal coefficients, give a particularly uniform distribution of the points throughout n-dimensional space and from their number theoretic properties minimize the error on a prescribed class of integrals. The method can be combined with the Monte Carlo procedure.

(d) Sag-Szekeres method

By transformation this method seeks to induce properties into the integrand which make it accurately integrable by the trapezoidal rule. The transformation also allows effective control over the number of integrand evaluations.

(e) Automatic adaptive procedures

An automatic adaptive strategy in several dimensions normally involves division of the region into subregions, concentrating the divisions in those parts of the region where the integrand is worst behaved. It is difficult to arrange with any generality for variable limits in the inner integral(s). For this reason, some methods use a region where all the limits are constants; this is called a hyper-rectangle. Integrals over regions defined by variable or infinite limits may be handled by transformation to a hyper-rectangle. Integrals over regions so irregular that such a transformation is not feasible may be handled by surrounding the region by an appropriate hyper-rectangle and defining the integrand to be zero outside the desired region. Such a technique should always be followed by a Monte Carlo method for integration.
The method used locally in each subregion produced by the adaptive subdivision process is usually one of three types: Monte Carlo, number theoretic or deterministic. Deterministic methods are usually the most rapidly convergent but are often expensive to use for high dimensionality and not as robust as the other techniques.

2.3. References

Comprehensive reference:


Special topics:


3. Recommendations on Choice and Use of Routines

The following three sub-sections consider in turn routines for: one-dimensional integrals over a finite interval, and over a semi-infinite or an infinite interval; and multi-dimensional integrals. Within each sub-section, routines are classified by the type of method, which ranges from simple rule evaluation to automatic adaptive algorithms. The recommendations apply particularly when the primary objective is simply to compute the value of one or more integrals, and in these cases the automatic adaptive routines are generally the most convenient and reliable, although also the most expensive in computing time.

Note however that in some circumstances it may be counter-productive to use an automatic routine. If the results of the quadrature are to be used in turn as input to a further computation (e.g. an 'outer' quadrature or an optimization
problem), then this further computation may be adversely affected by the 'jagged performance profile' of an automatic routine; a simple rule-evaluation routine may provide much better overall performance. For further guidance, the article Lyness [3] is recommended.

3.1. One-dimensional Integrals over a Finite Interval

(a) Integrand defined as a set of points

If \( f(x) \) is defined numerically at four or more points, then the Gill-Miller finite difference method (D01GAF) should be used. The interval of integration is taken to coincide with the range of \( x \)-values of the points supplied. It is in the nature of this problem that any routine may be unreliable. In order to check results independently and so as to provide an alternative technique the user may fit the integrand by Chebyshev series using E02ADF and then use routines E02AJF and E02AKF to evaluate its integral (which need not be restricted to the range of the integration points, as is the case for D01GAF). A further alternative is to fit a cubic spline to the data using E02BAF and then to evaluate its integral using E02BDF.

(b) Integrand defined as a function

If the functional form of \( f(x) \) is known, then one of the following approaches should be taken. They are arranged in the order from most specific to most general, hence the first applicable procedure in the list will be the most efficient. However, if the user does not wish to make any assumptions about the integrand, the most reliable routine to use will be D01AJF, although this will in general be less efficient for simple integrals.

(i) Rule-evaluation routines

If \( f(x) \) is known to be sufficiently well-behaved (more precisely, can be closely approximated by a polynomial of moderate degree), a Gaussian routine with a suitable number of abscissae may be used.

D01BBF may be used if it is required to examine the weights and abscissae. In this case, the user should write the code for the evaluation of quadrature summation (6).

(ii) Automatic adaptive routines

Firstly, several routines are available for integrands of the form \( w(x)g(x) \) where \( g(x) \) is a '
smooth' function (i.e., has no singularities, sharp peaks or violent oscillations in the interval of integration) and \( w(x) \) is a weight function of one of the following forms:

\[
(\alpha) (\beta) \ 
\begin{array}{c}
\text{k} \\
\text{l}
\end{array}
\]

\[
w(x) = (b-x)^k (x-a)^l \left( \log(b-x) \right)^k \left( \log(x-a) \right)^l
\]

where \( k, l = 0 \) or \( 1 \), \( (\alpha), (\beta) > -1 \): use D01APF;

if \( w(x) = 1/(x-c) \): use D01AQF (this integral is called the Hilbert transform of \( g \));

if \( w(x) = \cos((\omega)x) \) or \( \sin((\omega)x) \): use D01ANF (this routine can also handle certain types of singularities in \( g(x) \)).

Secondly, there are some routines for general \( f(x) \). If \( f(x) \) is known to be free of singularities, though it may be oscillatory, D01AKF may be used.

The most powerful of the finite interval integration routine is D01AJF (which can cope with singularities of several types). It may be used if none of the more specific situations described above applies. D01AJF is very reliable, particularly where the integrand has singularities other than at an end-point, or has discontinuities or cusps, and is therefore recommended where the integrand is known to be badly-behaved, or where its nature is completely unknown.

Most of the routines in this chapter require the user to supply a function or subroutine to evaluate the integrand at a single point.

If \( f(x) \) has singularities of certain types, discontinuities or sharp peaks occurring at known points, the integral should be evaluated separately over each of the subranges or D01ALF may be used.

3.2. One-dimensional Integrals over a Semi-infinite or Infinite Interval

(a) Integrand defined as a set of points

If \( f(x) \) is defined numerically at four or more points, and the portion of the integral lying outside the range of the points supplied may be neglected, then the Gill-Miller finite difference method, D01GAF, should be used.
(b) **Integrand defined as a function**

(i) **Rule evaluation routines**

If \( f(x) \) behaves approximately like a polynomial in \( x \),
apart from a weight function of the form
\[
-(\beta)x^e \quad (\beta)>0 \text{ (semi-infinite interval, lower limit finite)}; \\

\]

\[
-(\beta)x^e \quad (\beta)<0 \text{ (semi-infinite interval, upper limit finite)}; \\

\]

\[
2^{-(\beta)(x-(\alpha))} \quad (\beta)>0 \text{ (infinite interval)}; \\

\]

or if \( f(x) \) behaves approximately like a
\[
-1 \text{ (semi-infinite range); then the Gaussian routines may be used.} \\

D01BBF may be used if it is required to examine the weights and abscissae. In this case, the user should write the code for the evaluation of quadrature summation (6).

(ii) **Automatic adaptive routines**

D01AMF may be used, except for integrands which decay slowly towards an infinite end-point, and oscillate in sign over the entire range. For this class, it may be possible to calculate the integral by integrating between the zeros and invoking some extrapolation process.

D01ASF may be used for integrals involving weight functions of the form \( \cos((\omega)x) \) and \( \sin((\omega)x) \) over a semi-infinite interval (lower limit finite).

The following alternative procedures are mentioned for completeness, though their use will rarely be necessary:

1. If the integrand decays rapidly towards an infinite end-point, a finite cut-off may be chosen, and the finite range methods applied.

2. If the only irregularities occur in the finite part (apart from a singularity at the finite limit, with which D01AMF can cope), the range
may be divided, with D01AMF used on the infinite part.

(3) A transformation to finite range may be employed, e.g.

\[
1 - t \\
\frac{x}{t} = \quad \text{or} \quad x = -\log t
\]

will transform \((0, \infty)\) to \((1, 0)\) while for infinite ranges we have

\[
\int_{-\infty}^{\infty} f(x) \, dx = \int \left[f(x) + f(-x)\right] \, dx.
\]

If the integrand behaves badly on \((-\infty, 0)\) and well on \((0, \infty)\) or vice versa it is better to compute it as

\[
\int_{-\infty}^{0} f(x) \, dx + \int_{0}^{\infty} f(x) \, dx.
\]

This saves computing unnecessary function values in the semi-infinite range where the function is well behaved.

3.3. Multi-dimensional Integrals

A number of techniques are available in this area and the choice depends to a large extent on the dimension and the required accuracy. It can be advantageous to use more than one technique as a confirmation of accuracy particularly for high dimensional integrations. Many of the routines incorporate the transformation procedure REGION which allows general product regions to be easily dealt with in terms of conversion to the standard n-cube region.

(a) Products of one-dimensional rules (suitable for up to about 5 dimensions)

If \(f(x_1, x_2, \ldots, x_n)\) is known to be a sufficiently well-behaved function of each variable \(x_i\), apart possibly from \(x_i\) weight functions of the types provided, a product of Gaussian rules may be used. These are provided by D01BBF. In this case, the user should write the code for the evaluation of quadrature summation (6). Rules for finite, semi-infinite and infinite ranges are included.
The one-dimensional routines may also be used recursively. For example, the two-dimensional integral
\[
\int_{a}^{b} \int_{1}^{2} f(x,y) \, dy \, dx
\]
may be expressed as
\[
\int_{a}^{b} F(x) \, dx,
\]
where
\[
F(x) = \int_{a}^{2} f(x,y) \, dy.
\]
The user segment to evaluate \( F(x) \) will call the integration routine for the \( y \)-integration, which will call another user segment for \( f(x,y) \) as a function of \( y \) (\( x \) being effectively a constant). Note that, as Fortran is not a recursive language, a different library integration routine must be used for each dimension. Apart from this restriction, the full range of one-dimensional routines are available, for finite/infinite intervals, constant/variable limits, rule evaluation/automatic strategies etc.

(b) Automatic routines (D01GBF and D01FCF)

Both routines are for integrals of the form
\[
\int_{a}^{b} \int_{1}^{2} \ldots \int_{n-1}^{n} f(x_1, x_2, \ldots, x_n) \, dx_1 \, dx_2 \ldots \, dx_n.
\]
D01GBF is an adaptive Monte Carlo routine. This routine is usually slow and not recommended for high accuracy work. It is a robust routine that can often be used for low accuracy results with highly irregular
integrands or when \( n \) is large.

\texttt{D01FCF} is an adaptive deterministic routine. Convergence is fast for well-behaved integrands. Highly accurate results can often be obtained for \( n \) between 2 and 5, using significantly fewer integrand evaluations than would be required by \texttt{D01GBF}. The routine will usually work when the integrand is mildly singular and for \( n \leq 10 \) should be used before \texttt{D01GBF}. If it is known in advance that the integrand is highly irregular, it is best to compare results from at least two different routines.

There are many problems for which one or both of the routines will require large amounts of computing time to obtain even moderately accurate results. The amount of computing time is controlled by the number of integrand evaluations allowed by the user, and users should set this parameter carefully, with reference to the time available and the accuracy desired.

3.4. Decision Trees

(i) One-dimensional integrals over a finite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual

(ii) One-dimensional integrals over a semi-infinite or infinite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual

\texttt{D01 -- Quadrature} \hspace{1cm} \texttt{Contents -- D01}

Chapter D01

Quadrature

\texttt{D01AJF} \hspace{1cm} 1-D quadrature, adaptive, finite interval, strategy due to Piessens and de Doncker, allowing for badly-behaved integrands

\texttt{D01AKF} \hspace{1cm} 1-D quadrature, adaptive, finite interval, method suitable for oscillating functions
D01ALF 1-D quadrature, adaptive, finite interval, allowing for singularities at user-specified break-points

D01AMF 1-D quadrature, adaptive, infinite or semi-infinite interval

D01ANF 1-D quadrature, adaptive, finite interval, weight function cos((omega)x) or sin((omega)x)

D01APF 1-D quadrature, adaptive, finite interval, weight function with end-point singularities of algebraico-logarithmic type

D01AQF 1-D quadrature, adaptive, finite interval, weight function 1/(x-c), Cauchy principal value (Hilbert transform)

D01ASF 1-D quadrature, adaptive, semi-infinite interval, weight function cos((omega)x) or sin((omega)x)

D01BBF Weights and abscissae for Gaussian quadrature rules

D01FCF Multi-dimensional adaptive quadrature over hyper-rectangle

D01GAF 1-D quadrature, integration of function defined by data values, Gill-Miller method

D01GBF Multi-dimensional quadrature over hyper-rectangle, Monte Carlo method

D01AJF(3NAG) Foundation Library (12/10/92)  D01AJF(3NAG)

D01 -- Quadrature
    D01AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01AJF is a general-purpose integrator which calculates an approximation to the integral of a function f(x) over a finite interval [a,b]:

\[ I = \int_{a}^{b} f(x) \, dx \]

2. Specification

```
SUBROUTINE D01AJF (F, A, B, EPSABS, EPSREL, RESULT, 
1          ABSERR, W, LW, IW, LIW, IFAIL)
integer LW, IW(LIW), LIW, IFAIL
double precision F, A, B, EPSABS, EPSREL, RESULT, ABSERR, W
1 (LW)
external F
```

3. Description

D01AJF is based upon the QUADPACK routine QAGS (Piessens et al [3]). It is an adaptive routine, using the Gauss 10-point and Kronrod 21-point rules. The algorithm, described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The local error estimation is described by Piessens et al [3].

The routine is suitable as a general purpose integrator, and can be used when the integrand has singularities, especially when these are of algebraic or logarithmic type.

D01AJF requires the user to supply a function to evaluate the integrand at a single point.

The routine D01ATF(*) uses an identical algorithm but requires the user to supply a subroutine to evaluate the integrand at an array of points. Therefore D01ATF(*) will be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure
F must return the value of the integrand f at a given point.

Its specification is:

    DOUBLE PRECISION FUNCTION F (X)
    DOUBLE PRECISION X

1: X -- DOUBLE PRECISION Input
    On entry: the point at which the integrand f must be
    evaluated.
    F must be declared as EXTERNAL in the (sub)program from
    which D01AJF is called. Parameters denoted as Input
    must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input
    On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION Input
    On entry: the upper limit of integration, b. It is not
    necessary that a<b.

4: EPSABS -- DOUBLE PRECISION Input
    On entry: the absolute accuracy required. If EPSABS is
    negative, the absolute value is used. See Section 7.

5: EPSREL -- DOUBLE PRECISION Input
    On entry: the relative accuracy required. If EPSREL is
    negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION Output
    On exit: the approximation to the integral I.

7: ABSERR -- DOUBLE PRECISION Output
    On exit: an estimate of the modulus of the absolute error,
    which should be an upper bound for |I-RESULT|.

8: W(LW) -- DOUBLE PRECISION array Output
    On exit: details of the computation, as described in
    Section 8.

9: LW -- INTEGER Input
On entry:
The dimension of the array W as declared in the (sub)program from which D01AJF is called.
The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: LW >= 4.

10: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

11: LIW -- INTEGER Input
On entry:
The dimension of the array IW as declared in the (sub)program from which D01AJF is called.
The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a
discontinuity, etc) you will probably gain from splitting up
the interval at this point and calling the integrator on the
subranges. If necessary, another integrator, which is
designed for handling the type of difficulty involved, must
be used. Alternatively, consider relaxing the accuracy
requirements specified by EPSABS and EPSREL, or increasing
the amount of workspace.

IFAIL = 2
Round-off error prevents the requested tolerance from being
achieved. The error may be under-estimated. Consider
requesting less accuracy.

IFAIL = 3
Extremely bad local integrand behaviour causes a very strong
subdivision around one (or more) points of the interval.

IFAIL = 4
The requested tolerance cannot be achieved, because the
extrapolation does not increase the accuracy satisfactorily;
the returned result is the best which can be obtained. The
same advice applies as in the case of IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent.
Please note that divergence can occur with any non-zero
value of IFAIL.

IFAIL = 6
On entry LW < 4,
or        LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

\[ |\text{I-RESULT}| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{|\text{EPSABS}|,|\text{EPSREL}|*|\text{I}|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative
error tolerance. Moreover it returns the quantity ABSERR which,
in normal circumstances, satisfies

\[ |\text{I-RESULT}| \leq \text{ABSERR} \leq \text{tol}. \]
8. Further Comments

The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AJF along with the integral contributions and error estimates over the sub-intervals.

Specifically, for i=1,2,...,n, let \( r_i \) denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a, b]\) and \( e_i \) be the corresponding absolute error estimate.

\[
\int_{a}^{b} f(x)\,dx = \sum_{i=1}^{n} r_i \quad \text{and} \quad \text{RESULT} = \sum_{i=1}^{n} r_i, \quad \text{unless D01AJF terminates while testing for divergence of the integral (see Piessens et al [3], Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of n is returned in IW(1), and the values a_i, b_i, e_i and r_i are stored consecutively in the array W, that is:}
\]

\[
a_i = W(i), \\
b_i = W(n+i), \\
e_i = W(2n+i) \quad \text{and} \\
r_i = W(3n+i).
\]

9. Example

To compute

\[
\int_{0}^{\pi} x \sin(30x) \,dx.
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

D01AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01AKF is an adaptive integrator, especially suited to oscillating, non-singular integrands, which calculates an approximation to the integral of a function \( f(x) \) over a finite interval \([a,b]\):

\[
\int_{a}^{b} f(x) \, dx
\]

2. Specification

```fortran
SUBROUTINE D01AKF (F, A, B, EPSABS, EPSREL, RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
```

3. Description

D01AKF is based upon the QUADPACK routine QAG (Piessens et al [3]). It is an adaptive routine, using the Gauss 30-point and Kronrod 61-point rules. A 'global' acceptance criterion (as defined by Malcolm and Simpson [1]) is used. The local error estimation is described in Piessens et al [3].
Because this routine is based on integration rules of high order, it is especially suitable for non-singular oscillating integrands.

D01AKF requires the user to supply a function to evaluate the integrand at a single point.

The routine D01AUF(*) uses an identical algorithm but requires the user to supply a subroutine to evaluate the integrand at an array of points. Therefore D01AUF(*) will be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

D01AUF(*) also has an additional parameter KEY which allows the user to select from six different Gauss-Kronrod rules.

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure

F must return the value of the integrand f at a given point.

Its specification is:

\[ \text{DOUBLE PRECISION FUNCTION F (X)} \]
\[ \text{DOUBLE PRECISION X} \]

1: X -- DOUBLE PRECISION Input

On entry: the point at which the integrand f must be evaluated.

F must be declared as EXTERNAL in the (sub)program from which D01AKF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input

On entry: the lower limit of integration, a.
3: B -- DOUBLE PRECISION  
   On entry: the upper limit of integration, b. It is not necessary that a<b.

4: EPSABS -- DOUBLE PRECISION  
   On entry: the absolute accuracy required. If EPSABS is negative, the absolute value is used. See Section 7.

5: EPSREL -- DOUBLE PRECISION  
   On entry: the relative accuracy required. If EPSREL is negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION  
   On exit: the approximation to the integral I.

7: ABSERR -- DOUBLE PRECISION  
   On exit: an estimate of the modulus of the absolute error, which should be an upper bound |I-RESULT|.

8: W(LW) -- DOUBLE PRECISION array  
   On exit: details of the computation, as described in Section 8.

9: LW -- INTEGER  
   On entry: the dimension of W, as declared in the (sub)program from which D01AKF is called. The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: LW >= 4. See IW below.

10: IW(LIW) -- INTEGER array  
    On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

11: LIW -- INTEGER  
    On entry: the dimension of the array IW as declared in the (sub)program from which D01AKF is called. The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

12: IFAIL -- INTEGER  
    On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. Probably another integrator which is designed for handling the type of difficulty involved must be used. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
On entry LW < 4,

or  LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I\text{-RESULT}| \leq \text{tol}, \]

where
\[
tol = \max\{|\text{EPSABS}|, |\text{EPSREL}|*|I|\},
\]

and EPSABS and EPSREL are user-specified absolute and relative error tolerances. Moreover it returns the quantity ABSERR which, in normal circumstances satisfies

\[
|I - \text{RESULT}| \leq \text{ABSERR} \leq tol.
\]

8. Further Comments

The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL \neq 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AKF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for \( i=1,2,\ldots,n \), let \( r \) denote the approximation to \( i \) the value of the integral over the sub-interval \( [a_i, b_i] \) in the \( i \) partition of \( [a,b] \) and \( e \) be the corresponding absolute error estimate. Then, \( \int_a^b f(x)dx \approx r \) and \( \text{RESULT} = r \). The value of \( n \) and \( i \) is returned in \( \text{IW}(1) \), and the values \( a_i, b_i, e_i \) and \( r_i \) are stored consecutively in the array W, that is:

\[
a_i = W(i),
b_i = W(n+i),
e_i = W(2n+i) \text{ and } r_i = W(3n+i).
\]

9. Example

To compute
2(\pi) / | x \sin(30x) \cos x \, dx. / 0

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

D01ALF -- Quadrature

D01ALF is a general purpose integrator which calculates an approximation to the integral of a function \( f(x) \) over a finite interval \([a, b]\):

\[
\int_{a}^{b} f(x) \, dx
\]

where the integrand may have local singular behaviour at a finite number of points within the integration interval.

D01ALF is based upon the QUADPACK routine QAGP (Piessens et al
It is very similar to D01AJF, but allows the user to supply difficult. It is an adaptive routine, using the Gauss 10-point and Kronrod 21-point rules. The algorithm described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The user-supplied 'break-points' always occur as the end-points of some sub-interval during the adaptive process. The local error estimation is described by Piessens et al [3].

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.

   External Procedure
   F must return the value of the integrand f at a given point.

   Its specification is:

   DOUBLE PRECISION FUNCTION F (X)
   DOUBLE PRECISION X

   1: X -- DOUBLE PRECISION Input
      On entry: the point at which the integrand f must be evaluated.
      F must be declared as EXTERNAL in the (sub)program from
      which D01ALF is called. Parameters denoted as Input
      must not be changed by this procedure.

   2: A -- DOUBLE PRECISION Input
      On entry: the lower limit of integration, a.

   3: B -- DOUBLE PRECISION Input
      On entry: the upper limit of integration, b. It is not
necessary that \( a < b \).

4: NPTS -- INTEGER
   Input
   On entry: the number of user-supplied break-points within
   the integration interval. Constraint: NPTS >= 0.

5: POINTS(NPTS) -- DOUBLE PRECISION array
   Input
   On entry: the user-specified break-points. Constraint: the
   break-points must all lie within the interval of integration
   (but may be supplied in any order).

6: EPSABS -- DOUBLE PRECISION
   Input
   On entry: the absolute accuracy required. If EPSABS is
   negative, the absolute value is used. See Section 7.

7: EPSREL -- DOUBLE PRECISION
   Input
   On entry: the relative accuracy required. If EPSREL is
   negative, the absolute value is used. See Section 7.

8: RESULT -- DOUBLE PRECISION
   Input
   On entry: the approximation to the integral \( I \).

9: ABSERR -- DOUBLE PRECISION
   Output
   On exit: an estimate of the modulus of the absolute error,
   which should be an upper bound for \( |I - \text{RESULT}| \).

10: W(LW) -- DOUBLE PRECISION array
    Output
    On exit: details of the computation, as described in
    Section 8.

11: LW -- INTEGER
    Input
    On entry: the dimension of the array \( W \) as declared in the
    (sub)program from which D01ALF is called.
    The value of LW (together with that of LIW below) imposes a
    bound on the number of sub-intervals into which the interval
    of integration may be divided by the routine. The number of
    sub-intervals cannot exceed \((LW-2*NPTS-4)/4\). The more
difficult the integrand, the larger LW should be. Suggested
value: a value in the range 800 to 2000 is adequate for most

12: IW(LIW) -- INTEGER array
    Output
    On exit: IW(1) contains the actual number of sub-intervals
    used. The rest of the array is used as workspace.

13: LIW -- INTEGER
    Input
    On entry: the dimension of the array IW as declared in the
    (sub)program from which D01ALF is called.
The number of sub-intervals into which the interval of integration may be divided cannot exceed \((LIW-NPTS-2)/2\).
Suggested value: \(LIW = LW/2\). Constraint: \(LIW \geq NPTS + 4\).

14: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL \=/0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached, without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) it should be supplied to the routine as an element of the vector POINTS. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. The error may be under-estimated. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
The requested tolerance cannot be achieved, because the
extrapolation does not increase the accuracy satisfactorily; the result returned is the best which can be obtained. The same advice applies as in the case IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent. Please note that divergence can also occur with any other non-zero value of IFAIL.

IFAIL = 6
The input is invalid: break-points are specified outside the integration range, NPTS > LIMIT or NPTS < 0. RESULT and ABSERR are set to zero.

IFAIL = 7
On entry LW < 2*NPTS+8,
or LIW < NPTS+4.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I - \text{RESULT}| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{|\text{EPSABS}|,|\text{EPSREL}|*|I|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative error tolerances. Moreover it returns the quantity ABSERR which, in normal circumstances, satisfies

\[ |I - \text{RESULT}| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments
The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01ALF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a,b]\) and e be the corresponding absolute error
estimate. Then, \( \int f(x)\,dx \approx r \) and \( \text{RESULT} = r \) unless D01ALF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in IW(1), and the values \( a, b, e \) and \( r \) are stored consecutively in the array \( W \), that is:

\[
\begin{align*}
    a &= W(i), \\
    b &= W(n+i), \\
    e &= W(2n+i) \quad \text{and} \\
    r &= W(3n+i).
\end{align*}
\]

9. Example

To compute

\[
\int_{0}^{1} \frac{1}{\sqrt{|x-1/7|}} \, dx.
\]

A break-point is specified at \( x=1/7 \), at which point the integrand is infinite. (For definiteness the function FST returns the value 0.0 at this point.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

D01AMF calculates an approximation to the integral of a function \( f(x) \) over an infinite or semi-infinite interval \([a,b]\):

\[
\int_{a}^{b} f(x) \, dx
\]

2. Specification

```fortran
SUBROUTINE D01AMF (F, BOUND, INF, EPSABS, EPSREL, RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
INTEGER INF, LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION F, BOUND, EPSABS, EPSREL, RESULT, ABSERR, W(LW)
EXTERNAL F
```

3. Description

D01AMF is based on the QUADPACK routine QAGI (Piessens et al [3]) on \([0,1]\) using one of the identities:

\[
\int_{a}^{1} f(x) \, dx = \int_{0}^{1-\frac{1-t}{2}} f(\frac{a-1}{2} - \frac{1-t}{2}) \, dt \\
\int_{-\infty}^{a} f(x) \, dx = \int_{0}^{\frac{1-t}{2}} f(\frac{a+1}{2} - \frac{1-t}{2}) \, dt \\
\int_{-\infty}^{\infty} f(x) \, dx = \int_{0}^{\infty} (f(x)+f(-x)) \, dx = \int_{0}^{1} (\frac{f(-t)+f(t)}{2}) \, dt
\]
where $a$ represents a finite integration limit. An adaptive procedure, based on the Gauss seven-point and Kronrod 15-point rules, is then employed on the transformed integral. The algorithm, described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The local error estimation is described by Piessens et al [3].

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure

On entry: the point at which the integrand $f$ must be evaluated.

Its specification is:

$$\text{DOUBLE PRECISION FUNCTION } F (X)$$

DOUBLE PRECISION X

1: X -- DOUBLE PRECISION Input

On entry: the point at which the integrand $f$ must be evaluated.

$F$ must be declared as EXTERNAL in the (sub)program from which D01AMF is called. Parameters denoted as Input must not be changed by this procedure.

2: BOUND -- DOUBLE PRECISION Input

On entry: the finite limit of the integration range (if present). $BOUND$ is not used if the interval is doubly infinite.
3: INF -- INTEGER
   On entry: indicates the kind of integration range:
   if INF = 1, the range is \([\text{BOUND}, +\infty)\)
   if INF = -1, the range is \((-\infty, \text{BOUND}]\)
   if INF = +2, the range is \((-\infty, +\infty)\).
   Constraint: INF = -1, 1 or 2.

4: EPSABS -- DOUBLE PRECISION
   On entry: the absolute accuracy required. If EPSABS is
   negative, the absolute value is used. See Section 7.

5: EPSREL -- DOUBLE PRECISION
   On entry: the relative accuracy required. If EPSREL is
   negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION
   On exit: the approximation to the integral \(I\).

7: ABSERR -- DOUBLE PRECISION
   On exit: an estimate of the modulus of the absolute error,
   which should be an upper bound for \(|I - \text{RESULT}|\).

8: W(LW) -- DOUBLE PRECISION array
   On exit: details of the computation, as described in
   Section 8.

9: LW -- INTEGER
   On entry:
   the dimension of the array W as declared in the (sub)program
   from which D01AMF is called.
   The value of LW (together with that of LIW below) imposes a
   bound on the number of sub-intervals into which the interval
   of integration may be divided by the routine. The number of
   sub-intervals cannot exceed LW/4. The more difficult the
   integrand, the larger LW should be. Suggested value: a value
   in the range 800 to 2000 is adequate for most problems.
   Constraint: LW >= 4.

10: IW(LIW) -- INTEGER array
    On exit: IW(1) contains the actual number of sub-intervals
    used. The rest of the array is used as workspace.

11: LIW -- INTEGER
    On entry:
    the dimension of the array IW as declared in the
    (sub)program from which D01AMF is called.
    The number of sub-intervals into which the interval of
integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the requested accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) you will probably gain from splitting up the interval at this point and calling D01AMF on the infinite subrange and an appropriate integrator on the finite subrange. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. The error may be underestimated. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The
same advice applies as in the case of IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent. It must be noted that divergence can also occur with any other non-zero value of IFAIL.

IFAIL = 6
On entry LW < 4,
or LIW < 1,
or INF /= -1, 1 or 2.
Please note that divergence can occur with any non-zero value of IFAIL.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

$$|I - \text{RESULT}| \leq \text{tol},$$

where

$$\text{tol} = \max\{|\text{EPSABS}|, |\text{EPSREL}|*|I|\},$$

and EPSABS and EPSREL are user-specified absolute and relative error tolerances. Moreover it returns the quantity ABSERR, which, in normal circumstances, satisfies

$$|I - \text{RESULT}| \leq \text{ABSERR} \leq \text{tol}.$$  

8. Further Comments
The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AMF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r denote the approximation to the value of the integral over the sub-interval [a_i,b_i] in the partition of [a,b] and e_i be the corresponding absolute error.
estimate. Then, \(| f(x)dx - \approx r \) and RESULT\( \approx r \) unless D01AMF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in IW(1), and the values \( a, b, e \) and \( r \) are stored consecutively in the array \( W \), that is:

\[
\begin{align*}
\quad a &= W(i), b = W(n+i), e = W(2n+i) \quad \text{and} \quad r = W(3n+i).
\end{align*}
\]

Note: that this information applies to the integral transformed to \((0,1)\) as described in Section 3, not to the original integral.

9. Example

To compute

\[
\int_0^{\infty} \frac{1}{x(x+1)} dx.
\]

The exact answer is \( (\pi) \).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D01ANF calculates an approximation to the sine or the cosine transform of a function g over [a,b]:

\[
I = \int_a^b |g(x)| \sin(\omega x) \, dx \quad \text{or} \quad I = \int_a^b |g(x)| \cos(\omega x) \, dx
\]

(for a user-specified value of (omega)).

2. Specification

```fortran
SUBROUTINE D01ANF (G, A, B, OMEGA, KEY, EPSABS, EPSREL, RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
  INTEGER KEY, LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION G, A, B, OMEGA, EPSABS, EPSREL, RESULT, ABSERR, W(LW)
  EXTERNAL G
```

3. Description

D01ANF is based upon the QUADPACK routine QFOUR (Piessens et al. [3]). It is an adaptive routine, designed to integrate a function of the form \( g(x)w(x) \), where \( w(x) \) is either \( \sin(\omega x) \) or \( \cos(\omega x) \). If a sub-interval has length

\[
L = \frac{|b-a|}{2}
\]

then the integration over this sub-interval is performed by means of a modified Clenshaw-Curtis procedure (Piessens and Branders [2]) if \( L(\omega) > 4 \) and \( 1 \leq L \leq 20 \). In this case a Chebyshev-series approximation of degree 24 is used to approximate \( g(x) \), while an error estimate is computed from this approximation together with that obtained using Chebyshev-series of degree 12. If the above conditions do not hold then Gauss 7-point and Kronrod 15-point rules are used. The algorithm, described in [3], incorporates a global acceptance criterion (as defined in Malcolm and Simpson [1]) together with the (epsilon)-algorithm Wynn [4] to perform extrapolation. The local error estimation is described in [3].

4. References


5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
   External Procedure
   G must return the value of the function g at a given point.
   Its specification is:

   DOUBLE PRECISION FUNCTION G (X)
   DOUBLE PRECISION X

   1: X -- DOUBLE PRECISION
      Input
      On entry: the point at which the function g must be
      evaluated.
      G must be declared as EXTERNAL in the (sub)program from
      which D01ANF is called. Parameters denoted as Input
      must not be changed by this procedure.

2: A -- DOUBLE PRECISION
   Input
   On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION
   Input
   On entry: the upper limit of integration, b. It is not
   necessary that a<b.

4: OMEGA -- DOUBLE PRECISION
   Input
   On entry: the parameter (omega) in the weight function of
   the transform.

5: KEY -- INTEGER
   Input
   On entry: indicates which integral is to be computed:
   if KEY = 1, \( w(x) = \cos((\omega)x) \);
   if KEY = 2, \( w(x) = \sin((\omega)x) \).
   Constraint: KEY = 1 or 2.

6: EPSABS -- DOUBLE PRECISION
   Input
   On entry: the absolute accuracy required. If EPSABS is
   negative, the absolute value is used. See Section 7.

7: EPSREL -- DOUBLE PRECISION
   Input

   (1983) QUADPACK, A Subroutine Package for Automatic
   Integration. Springer-Verlag.

[4] Wynn P (1956) On a Device for Computing the e \( (S) \)
On entry: the relative accuracy required. If EPSREL is negative, the absolute value is used. See Section 7.

8: RESULT -- DOUBLE PRECISION
   On exit: the approximation to the integral I.

9: ABSERR -- DOUBLE PRECISION
   On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

10: W(LW) -- DOUBLE PRECISION array
    On exit: details of the computation, as described in Section 8.

11: LW -- INTEGER
    On entry:
    the dimension of the array W as declared in the (sub)program from which D01ANF is called.
    The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: LW >= 4.

12: IW(LIW) -- INTEGER array
    On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

13: LIW -- INTEGER
    On entry:
    the dimension of the array IW as declared in the (sub)program from which D01ANF is called.
    The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW/2. Suggested value: LIW = LW/2. Constraint: LIW >= 2.

14: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requested being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) you will probably gain from splitting up the interval at this point and calling the integrator on the subranges. If necessary, another integrator, which is designed for handling the type of difficulty involved, must be used. Alternatively consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. The error may be underestimated. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local behaviour of g(x) causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
The requested tolerance cannot be achieved because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The same advice applies as in the case of IFAIL = 1.

IFAIL= 5
The integral is probably divergent, or slowly convergent. It must be noted that divergence can occur with any non-zero value of IFAIL.

IFAIL= 6
On entry KEY < 1,

or

KEY > 2.

IFAIL= 7
On entry LW < 4,
or LIW < 2.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I-RESULT| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{|\text{EPSABS}|, |\text{EPSREL}| \times |I|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative tolerances. Moreover it returns the quantity ABSERR, which, in normal circumstances, satisfies

\[ |I-RESULT| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments

The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01ANF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for \( i = 1,2,...,n \), let \( r \) denote the approximation to the value of the integral over the sub-interval \( [a_i, b_i] \) in the partition of \( [a,b] \) and \( e \) be the corresponding absolute error estimate. Then, \( \int_a^b g(x)w(x)dx = r \) and RESULT = \( r \) unless D01ANF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in IW(1), and the values \( a_i, b_i, e_i, r_i \) are stored consecutively in the arrays.
array \ W, \text{ that is:}

\begin{align*}
a &= W(i), \\
b &= W(n+i), \\
e &= W(2n+i) \text{ and} \\
r &= W(3n+i).
\end{align*}

9. Example

To compute

\[
\frac{1}{\sqrt{\ln(x)\sin(10\pi x)}} dx.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**D01APF (3NAG)**

_D01APF -- NAG Foundation Library Routine Document_

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

_D01APF is an adaptive integrator which calculates an approximation to the integral of a function \( g(x)w(x) \) over a finite interval \([a,b]\):_

\[
\frac{1}{\sqrt{\ln(x)\sin(10\pi x)}} dx
\]
where the weight function $w$ has end-point singularities of algebraico-logarithmic type.

2. Specification

```fortran
SUBROUTINE D01APF (G, A, B, ALFA, BETA, KEY, EPSABS,
  1           EPSREL, RESULT, ABSERR, W, LW, IW, LIW,
  2           IFAIL)
  INTEGER KEY, LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION G, A, B, ALFA, BETA, EPSABS, EPSREL,
  1           RESULT, ABSERR, W(LW)
  EXTERNAL G
```

3. Description

D01APF is based upon the QUADPACK routine QAWSE (Piessens et al [3]) and integrates a function of the form $g(x)w(x)$, where the weight function $w(x)$ may have algebraico-logarithmic singularities at the end-points $a$ and/or $b$. The strategy is a modification of that in D01AKF. We start by bisecting the original interval and applying modified Clenshaw-Curtis integration of orders 12 and 24 to both halves. Clenshaw-Curtis integration is then used on all sub-intervals which have $a$ or $b$ as one of their end-points (Piessens et al [2]). On the other sub-intervals Gauss-Kronrod (7-15 point) integration is carried out.

A 'global' acceptance criterion (as defined by Malcolm and Simpson [1]) is used. The local error estimation control is described by Piessens et al [3].

4. References


5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
G must return the value of the function g at a given point X.

Its specification is:

```fortran
DOUBLE PRECISION FUNCTION G (X)
DOUBLE PRECISION X

1: X -- DOUBLE PRECISION Input
   On entry: the point at which the function g must be evaluated.
   G must be declared as EXTERNAL in the (sub)program from which D01APF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input
   On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION Input
   On entry: the upper limit of integration, b. Constraint: B > A.

4: ALFA -- DOUBLE PRECISION Input
   On entry: the parameter (alpha) in the weight function.
   Constraint: ALFA > -1.

5: BETA -- DOUBLE PRECISION Input
   On entry: the parameter (beta) in the weight function.
   Constraint: BETA > -1.

6: KEY -- INTEGER Input
   On entry: indicates which weight function is to be used:
   (alpha) (beta)
   if KEY = 1, w(x)=(x-a) (b-x)

   (alpha) (beta)
   if KEY = 2, w(x)=(x-a) (b-x) ln(x-a)

   (alpha) (beta)
   if KEY = 3, w(x)=(x-a) (b-x) ln(b-x)

   (alpha) (beta)
   if KEY = 4, w(x)=(x-a) (b-x) ln(x-a)ln(b-x)

   Constraint: KEY = 1, 2, 3 or 4

7: EPSABS -- DOUBLE PRECISION Input
   On entry: the absolute accuracy required. If EPSABS is negative, the absolute value is used. See Section 7.
8: EPSREL -- DOUBLE PRECISION  
   On entry: the relative accuracy required. If EPSREL is 
   negative, the absolute value is used. See Section 7.

9: RESULT -- DOUBLE PRECISION  
   On exit: the approximation to the integral I.

10: ABSERR -- DOUBLE PRECISION  
    On exit: an estimate of the modulus of the absolute error, 
    which should be an upper bound for |I-RESULT|.

11: W(LW) -- DOUBLE PRECISION array  
    On exit: details of the computation, as described in 
    Section 8.

12: LW -- INTEGER  
    On entry: 
    the dimension of the array W as declared in the (sub)program 
    from which D01APF is called. 
    The value of LW (together with that of LIW below) imposes a 
    bound on the number of sub-intervals into which the interval 
    of integration may be divided by the routine. The number of 
    sub-intervals cannot exceed LW/4. The more difficult the 
    integrand, the larger LW should be. Suggested value: LW = 
    800 to 2000 is adequate for most problems. Constraint: LW >= 
    8.

13: IW(LIW) -- INTEGER array  
    On exit: IW(1) contains the actual number of sub-intervals 
    used. The rest of the array is used as workspace.

14: LIW -- INTEGER  
    On entry: 
    the dimension of the array IW as declared in the 
    (sub)program from which D01APF is called. 
    The number of sub-intervals into which the interval of 
    integration may be divided cannot exceed LIW. Suggested 

15: IFAIL -- INTEGER  
    On entry: IFAIL must be set to 0, -1 or 1. Users who are 
    unfamiliar with this parameter should refer to the Essential 
    Introduction for details.

    On exit: IFAIL = 0 unless the routine detects an error or 
    gives a warning (see Section 6).

For this routine, because the values of output parameters 
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a discontinuity or a singularity of algebraico-logarithmic type within the interval can be determined, the interval must be split up at this point and the integrator called on the subranges. If necessary, another integrator, which is designed for handling the difficulty involved, must be used. Alternatively consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL = 2
Round-off error prevents the requested tolerance from being achieved. Consider requesting less accuracy.

IFAIL = 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
On entry B <= A,
or ALFA <= -1,
or BETA <= -1,
or KEY < 1,
or KEY > 4.

IFAIL = 5
On entry LW < 8,
or LIW < 2.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I - \text{RESULT}| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{|\text{EPSABS}|, |\text{EPSREL}| \times |I|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative error tolerances.

Moreover it returns the quantity ABSERR which, in normal circumstances, satisfies:

\[ |I - \text{RESULT}| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments

The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01APF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for \( i=1,2,...,n \), let \( r \) denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a, b]\) and \( e \) be the corresponding absolute error estimate. Then, \( \int_a^b f(x)w(x)dx = r \) and \( \text{RESULT} > r \). The value of \( \int_a^b f(x)w(x)dx \) is returned in IW(1), and the values \( a, b, e \) and \( r \) are stored consecutively in the array W, that is:

\[ a = W(i), \quad i \]
\[ b = W(n+i), \quad i \]
9. Example
To compute:

\[
\int_{1}^{\infty} \ln(x) \cos(10\pi x) \, dx \quad \text{and} \quad \int_{0}^{1} \frac{1}{\sin(10x)} \, dx.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
for user-specified values of a, b and c.

2. Specification

```fortran
SUBROUTINE D01AQF (G, A, B, C, EPSABS, EPSREL, RESULT,
1                      ABSERR, W, LW, IW, LIW, IFAIL)
INTEGER LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION G, A, B, C, EPSABS, EPSREL, RESULT,
1                      ABSERR, W(LW)
EXTERNAL G
```

3. Description

D01AQF is based upon the QUADPACK routine QAWC (Piessens et al [3]) and integrates a function of the form $g(x)w(x)$, where the weight function

$$w(x) = \frac{1}{x-c}$$

is that of the Hilbert transform. (If $a<c<b$ the integral has to be interpreted in the sense of a Cauchy principal value.) It is an adaptive routine which employs a 'global' acceptance criterion (as defined by Malcolm and Simpson [1]). Special care is taken to ensure that $c$ is never the end-point of a sub-interval (Piessens et al [2]). On each sub-interval $(c_1, c_2)$ modified Clenshaw-Curtis integration of orders 12 and 24 is performed if $|c_1-c| \leq |c_2-c|$ where $d=(c_2-c_1)/20$. Otherwise the Gauss 7-point and Kronrod 15-point rules are used. The local error estimation is described by Piessens et al [3].

4. References


5. Parameters
1: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
   External Procedure
   G must return the value of the function g at a given point.

   Its specification is:

   DOUBLE PRECISION FUNCTION G (X)
   DOUBLE PRECISION X

   1: X -- DOUBLE PRECISION
      On entry: the point at which the function g must be
      evaluated.
      G must be declared as EXTERNAL in the (sub)program from
      which D01AQF is called. Parameters denoted as Input
      must not be changed by this procedure.

2: A -- DOUBLE PRECISION
   On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION
   On entry: the upper limit of integration, b. It is not
   necessary that a<b.

4: C -- DOUBLE PRECISION
   On entry: the parameter c in the weight function.
   Constraint: C must not equal A or B.

5: EPSABS -- DOUBLE PRECISION
   On entry: the absolute accuracy required. If EPSABS is
   negative, the absolute value is used. See Section 7.

6: EPSREL -- DOUBLE PRECISION
   On entry: the relative accuracy required. If EPSREL is
   negative, the absolute value is used. See Section 7.

7: RESULT -- DOUBLE PRECISION
   On exit: the approximation to the integral I.

8: ABSERR -- DOUBLE PRECISION
   On exit: an estimate of the modulus of the absolute error,
   which should be an upper bound for |I-RESULT|.

9: W(LW) -- DOUBLE PRECISION array
   On exit: details of the computation, as described in
   Section 8.

10: LW -- INTEGER
    On entry:
    the dimension of the array W as declared in the (sub)program
from which D01AQF is called.
The value of LW (together with that of LIW below) imposes a
bound on the number of sub-intervals into which the interval
of integration may be divided by the routine. The number of
sub-intervals cannot exceed LW/4. The more difficult the
integrand, the larger LW should be. Suggested value: LW =
800 to 2000 is adequate for most problems. Constraint: LW >=
4.

11: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals
used. The rest of the array is used as workspace.

12: LIW -- INTEGER Input
On entry:
the dimension of the array IW as declared in the
(sub)program from which D01AQF is called.
The number of sub-intervals into which the interval of
integration may be divided cannot exceed LIW. Suggested

13: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /= 0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given
workspace has been reached without the accuracy requirements
being achieved. Look at the integrand in order to determine
the integration difficulties. Another integrator which is
designed for handling the type of difficulty involved, must
be used. Alternatively consider relaxing the accuracy
requirements specified by EPSABS and EPSREL, or increasing
the workspace.
IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local behaviour of g(x) causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
On entry C = A or C = B.

IFAIL= 5
On entry LW < 4, or LIW < 1.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I-RESULT| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{\text{EPSABS}, |\text{EPSREL}| \times |I| \}, \]

and EPSABS and EPSREL are user-specified absolute and relative error tolerances. Moreover it returns the quantity ABSERR which, in normal circumstances satisfies:

\[ |I-RESULT| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments
The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AQF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let \( r_i \) denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a,b]\) and \( e_i \) be the corresponding absolute error
CHAPTER 15. CHAPTER N

\[ \int_{a}^{b} g(x)w(x)dx = \text{estimate.} \]
\[ \text{RESULT} = r. \]

The value of \( \int_{a}^{b} g(x)w(x)dx \) is returned in \( \text{IW(1)} \), and the values \( a, b, e, \) and \( r \) are stored consecutively in the array \( W \), that is:

\[ a = W(i), \]
\[ b = W(n+i), \]
\[ e = W(2n+i), \]
\[ r = W(3n+i). \]

9. Example

To compute the Cauchy principal value of

\[ \frac{1}{2} \int_{-1}^{1} \frac{dx}{(x + 0.01)^2 (x - 0)} \]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

D01ASF calculates an approximation to the sine or the cosine transform of a function \( g \) over \([a, \infty)\):

\[
I = \int_{\infty}^{a} g(x) \sin(\omega x) \, dx \quad \text{or} \quad I = \int_{a}^{\infty} g(x) \cos(\omega x) \, dx
\]

(for a user-specified value of \( \omega \)).

2. Specification

SUBROUTINE D01ASF (G, A, OMEGA, KEY, EPSABS, RESULT, 1 ABSERR, LIMLST, LST, ERLST, RSLST, 2 IERLST, W, LW, IW, LIW, IFAIL)

INTEGER KEY, LIMLST, LST, IERLST(LIMLST), LW, IW 1 (LIW), LIW, IFAIL

DOUBLE PRECISION G, A, OMEGA, EPSABS, RESULT, ABSERR, ERLST 1 (LIMLST), RSLST(LIMLST), W(LW)

EXTERNAL G

3. Description

D01ASF is based upon the QUADPACK routine QAWFE (Piessens et al [2]). It is an adaptive routine, designed to integrate a function of the form \( g(x)w(x) \) over a semi-infinite interval, where \( w(x) \) is either \( \sin(\omega x) \) or \( \cos(\omega x) \). Over successive intervals

\[
C = [a + (k-1)c, a + kc], \quad k = 1, 2, \ldots, \text{LST}
\]

integration is performed by the same algorithm as is used by D01ANF. The intervals \( C \) are of constant length

\[
c = 2\{2[(\omega)] + 1\}(\pi)/|\omega|, \quad (\omega) \neq 0
\]

where \([\omega]\) represents the largest integer less than or equal to \( |\omega| \). Since \( c \) equals an odd number of half periods, the integral contributions over succeeding intervals will
alternate in sign when the function g is positive and
monotonically decreasing over \([a, \infty)\). The algorithm, described
by \[2\], incorporates a global acceptance criterion (as defined by
Malcolm and Simpson \[1\]) together with the \((\epsilon)\)-algorithm
(Wynn \[3\]) to perform extrapolation. The local error estimation
is described by Piessens et al \[2\].

If \((\omega) = 0\) and \(KEY = 1\), the routine uses the same algorithm as
D01AMF (with \(EPSREL = 0.0\)).

In contrast to the other routines in Chapter D01, D01ASF works
only with a user-specified absolute error tolerance \((EPSABS)\).
Over the interval \(C\) it attempts to satisfy the absolute accuracy
\(k\) requirement
\[
EPSA = U \cdot EPSABS
\]
\[k \quad k\]
\[k-1\]
where \(U = (1-p)p^k\), for \(k=1,2,...\) and \(p=0.9\).

However, when difficulties occur during the integration over the
\(k\)th sub-interval \(C\) such that the error flag \(IERLST(k)\) is non-
\(k\) zero, the accuracy requirement over subsequent intervals is
relaxed. See Piessens et al \[2\] for more details.

4. References

1 129--146.

(1983) QUADPACK, A Subroutine Package for Automatic
Integration. Springer-Verlag.

[3] Wynn P (1956) On a Device for Computing the \(e^{S_m n}\)

5. Parameters

1: \(G \quad -- \quad DOUBLE\ PRECISION\ FUNCTION,\ supplied\ by\ the\ user.\)
\Large External Procedure\ 
\(G\) must return the value of the function \(g\) at a given point.

Its specification is:

\begin{verbatim}
DOUBLE PRECISION FUNCTION G (X)
\end{verbatim}
DOUBLE PRECISION X

1: X -- DOUBLE PRECISION Input
   On entry: the point at which the function g must be evaluated.
   G must be declared as EXTERNAL in the (sub)program from which D01ASF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input
   On entry: the lower limit of integration, a.

3: OMEGA -- DOUBLE PRECISION Input
   On entry: the parameter (omega) in the weight function of the transform.

4: KEY -- INTEGER Input
   On entry: indicates which integral is to be computed:
   if KEY = 1, w(x)=cos((omega)x);
   if KEY = 2, w(x)=sin((omega)x).
   Constraint: KEY = 1 or 2.

5: EPSABS -- DOUBLE PRECISION Input
   On entry: the absolute accuracy requirement. If EPSABS is negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION Output
   On exit: the approximation to the integral I.

7: ABSERR -- DOUBLE PRECISION Output
   On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

8: LIMLST -- INTEGER Input
   On entry: an upper bound on the number of intervals C_k needed for the integration. Suggested value: LIMLST = 50 is adequate for most problems. Constraint: LIMLST >= 3.

9: LST -- INTEGER Output
   On exit: the number of intervals C_k actually used for the integration.

10: ERLST(LIMLST) -- DOUBLE PRECISION array Output
    On exit: ERLST(k) contains the error estimate corresponding to the integral contribution over the interval C_k for k=1,2,...,LST.
11: RSLST(LMLST) -- DOUBLE PRECISION array
   Output
   On exit: RSLST(k) contains the integral contribution over
   the interval C for k=1,2,...,LST.

12: IERLST(LMLST) -- INTEGER array
    Output
    On exit: IERLST(k) contains the error flag corresponding to
    RSLST(k), for k=1,2,...,LST. See Section 6.

13: W(LW) -- DOUBLE PRECISION array
    Workspace

14: LW -- INTEGER
    Input
    On entry:
     the dimension of the array W as declared in the (sub)program
     from which D01ASF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which each
     interval C may be divided by the routine. The number of
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: a value
     in the range 800 to 2000 is adequate for most problems.
     Constraint: LW >= 4.

15: IW(LIW) -- INTEGER array
    Output
    On exit: IW(1) contains the maximum number of sub-intervals
    actually used for integrating over any of the intervals C .
    The rest of the array is used as workspace.

16: LIW -- INTEGER
    Input
    On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01ASF is called.
     The number of sub-intervals into which each interval C may
     be divided cannot exceed LIW/2. Suggested value: LIW = LW/2.
     Constraint: LIW >= 2.

17: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).

    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) you will probably gain from splitting up the interval at this point and calling D01ASF on the infinite subrange and an appropriate integrator on the finite subrange. Alternatively, consider relaxing the accuracy requirements specified by EPSABS or increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. The error may be underestimated. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The same advice applies as in the case of IFAIL = 1.

IFAIL= 5
The integral is probably divergent, or slowly convergent. Please note that divergence can occur with any non-zero value of IFAIL.

IFAIL= 6
On entry KEY < 1,

or KEY > 2,
or $\text{LIMLST} < 3$.

**IFAIL= 7**
Bad integration behaviour occurs within one or more of the intervals $C_k$. Location and type of the difficulty involved can be determined from the vector $\text{IERLST}$ (see below).

**IFAIL= 8**
Maximum number of intervals $C_k$ ($= \text{LIMLST}$) allowed has been achieved. Increase the value of $\text{LIMLST}$ to allow more cycles.

**IFAIL= 9**
The extrapolation table constructed for convergence acceleration of the series formed by the integral contribution over the intervals $C_k$, does not converge to the required accuracy.

**IFAIL= 10**
On entry $\text{LW} < 4$,  
or $\text{LIW} < 2$.

In the cases $\text{IFAIL} = 7$, 8 or 9, additional information about the cause of the error can be obtained from the array $\text{IERLST}$, as follows:

$\text{IERLST}(k)=1$
The maximum number of subdivisions $= \min(\text{LW}/4,\text{LIW}/2)$ has been achieved on the $k$th interval.

$\text{IERLST}(k)=2$
Occurrence of round-off error is detected and prevents the tolerance imposed on the $k$th interval from being achieved.

$\text{IERLST}(k)=3$
Extremely bad integrand behaviour occurs at some points of the $k$th interval.

$\text{IERLST}(k)=4$
The integration procedure over the $k$th interval does not converge (to within the required accuracy) due to round-off in the extrapolation procedure invoked on this interval. It is assumed that the result on this interval is the best which can be obtained.

$\text{IERLST}(k)=5$
The integral over the $k$th interval is probably divergent or
slowly convergent. It must be noted that divergence can occur with any other value of \( \text{IERLST}(k) \).

7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I-\text{RESULT}| \leq |\text{EPSABS}|, \]

where \( \text{EPSABS} \) is the user-specified absolute error tolerance. Moreover, it returns the quantity \( \text{ABSERR} \), which, in normal circumstances, satisfies

\[ |I-\text{RESULT}| \leq \text{ABSERR} \leq |\text{EPSABS}|. \]

8. Further Comments

None.

9. Example

To compute

\[
\int_{0}^{\infty} \frac{\cos((\pi)x/2)}{x} \, dx.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.
1. Purpose

D01BBF returns the weights and abscissae appropriate to a Gaussian quadrature formula with a specified number of abscissae. The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-Laguerre and Gauss-Hermite.

2. Specification

```
SUBROUTINE D01BBF (A, B, ITYPE, N, WEIGHT, ABSCIS, GTYPE, IFAIL)
    INTEGER ITYPE, N, GTYPE, IFAIL
    DOUBLE PRECISION A, B, WEIGHT(N), ABSCIS(N)
```

3. Description

This routine returns the weights and abscissae for use in the Gaussian quadrature of a function $f(x)$. The quadrature takes the form

$$\sum_{i=1}^{n} w_i f(x_i)$$

where $w_i$ are the weights and $x_i$ are the abscissae (see Davis and Rabinowitz [1], Froberg [2], Ralston [3] or Stroud and Secrest [4]).

Weights and abscissae are available for Gauss-Legendre, Gauss-Rational, Gauss-Laguerre and Gauss-Hermite quadrature, and for a selection of values of $n$ (see Section 5).

(a) Gauss-Legendre Quadrature:

$$\int_{a}^{b} f(x) dx = \sum_{i=1}^{n} w_i f(x_i)$$

where $a$ and $b$ are finite and it will be exact for any function of the form

$$\sum_{i=0}^{2n-1} c_i x^i$$
(b) Gauss-Rational quadrature:
\[
\int_{a}^{\infty} f(x) \, dx \quad (a+b>0) \quad \text{or} \quad \int_{-\infty}^{a} f(x) \, dx \quad (a+b<0)
\]
and will be exact for any function of the form
\[
2n-1 \quad \sum_{i=0}^{2n-1} c \frac{1}{(x+b)}^i
\]
\[
f(x) = \sum_{i=0}^{2n+1} \frac{1}{(x+b)}^i
\]

(c) Gauss-Laguerre quadrature, adjusted weights option:
\[
\int_{a}^{\infty} f(x) \, dx \quad (b>0) \quad \text{or} \quad \int_{-\infty}^{a} f(x) \, dx \quad (b<0)
\]
and will be exact for any function of the form
\[
2n-1 \quad \sum_{i=0}^{2n-1} c \frac{1}{e^{-bx}(x+b)}^i
\]
\[
f(x) = e^{-bx} \sum_{i=0}^{2n+1} \frac{1}{e^{-bx}(x+b)}^i
\]

(d) Gauss-Hermite quadrature, adjusted weights option:
\[
\int_{-\infty}^{\infty} f(x) \, dx
\]
and will be exact for any function of the form
\[
2n-1 \quad \sum_{i=0}^{2n-1} c \frac{1}{e^{-b(x-a)}}^i
\]
\[
f(x) = e^{-b(x-a)} \sum_{i=0}^{2n+1} \frac{1}{e^{-b(x-a)}}^i
\]

(e) Gauss-Laguerre quadrature, normal weights option:
\[
\int_{a}^{\infty} e f(x) \, dx \quad (b>0) \quad \text{or} \quad \int_{-\infty}^{a} e f(x) \, dx
\]
and will be exact for any function of the form
\[
2n-1 \quad \sum_{i=0}^{2n-1} c \frac{e^{-bx}}{e^{-bx}(x+b)}^i
\]
\[
f(x) = e^{-bx} \sum_{i=0}^{2n+1} \frac{1}{e^{-bx}(x+b)}^i
\]
and will be exact for any function of the form
\[ \sum_{i=0}^{2n-1} c_i x^i \]

(f) Gauss-Hermite quadrature, normal weights option:
\[
\int_{-\infty}^{+\infty} e^{-b(x-a)^2} f(x) dx
\]

and will be exact for any function of the form:
\[ \sum_{i=0}^{2n-1} c_i x^i \]

Note: that the Gauss-Legendre abscissae, with \(a=-1, b=+1\), are the zeros of the Legendre polynomials; the Gauss-Laguerre abscissae, with \(a=0, b=1\), are the zeros of the Laguerre polynomials; and the Gauss-Hermite abscissae, with \(a=0, b=1\), are the zeros of the Hermite polynomials.

4. References


5. Parameters

1: A -- DOUBLE PRECISION
2: B -- DOUBLE PRECISION
   On entry: the quantities a and b as described in the
   appropriate subsection of Section 3.

3: ITYPE -- INTEGER
   On entry: indicates the type of weights for Gauss-Laguerre
   or Gauss-Hermite quadrature (see Section 3):
   if ITYPE = 1, adjusted weights will be returned;
   if ITYPE = 0, normal weights will be returned.
   Constraint: ITYPE = 0 or 1.
   For Gauss-Legendre or Gauss-Rational quadrature, this
   parameter is not used.

4: N -- INTEGER
   On entry: the number of weights and abscissae to be
   returned, n. Constraint: N = 1,2,3,4,5,6,8,10,12,14,16,20,24,32,48 or 64.

5: WEIGHT(N) -- DOUBLE PRECISION array
   On exit: the N weights. For Gauss-Laguerre and Gauss-
   Hermite quadrature, these will be the adjusted weights if
   ITYPE = 1, and the normal weights if ITYPE = 0.

6: ABSCIS(N) -- DOUBLE PRECISION array
   On exit: the N abscissae.

7: GTYPE -- INTEGER
   On entry: The value of GTYPE indicates which quadrature
   formula are to be used:
   GTYPE = 0 for Gauss-Legendre weights and abscissae;
   GTYPE = 1 for Gauss-Rational weights and abscissae;
   GTYPE = 2 for Gauss-Laguerre weights and abscissae;
   GTYPE = 3 for Gauss-Hermite weights and abscissae.

8: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
The N-point rule is not among those stored. If the soft fail option is used, the weights and abscissae returned will be those for the largest valid value of N less than the requested value, and the excess elements of WEIGHT and ABSCIS (i.e., up to the requested N) will be filled with zeros.

IFAIL= 2
The value of A and/or B is invalid.
  Gauss-Rational: $A + B = 0$
  Gauss-Laguerre: $B = 0$
  Gauss-Hermite: $B \leq 0$
If the soft fail option is used the weights and abscissae are returned as zero.

IFAIL= 3
Laguerre and Hermite normal weights only: underflow is occurring in evaluating one or more of the normal weights. If the soft fail option is used, the underflowing weights are returned as zero. A smaller value of N must be used; or adjusted weights should be used (ITYPE = 1). In the latter case, take care that underflow does not occur when evaluating the integrand appropriate for adjusted weights.

IFAIL= 4
  GTYPE < 0 or GTYPE > 3

7. Accuracy
The weights and abscissae are stored for standard values of A and B to full machine accuracy.

8. Further Comments
Timing is negligible.

9. Example
This example program returns the abscissae and (adjusted) weights for the six-point Gauss-Laguerre formula.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01FCF attempts to evaluate a multi-dimensional integral (up to 15 dimensions), with constant and finite limits, to a specified relative accuracy, using an adaptive subdivision strategy.

2. Specification

```fortran
SUBROUTINE D01FCF (NDIM, A, B, MINPTS, MAXPTS, FUNCTN,
  1 EPS, ACC, LENWRK, WRKSTR, FINVAL, IFAIL)
INTEGER NDIM, MINPTS, MAXPTS, LENWRK, IFAIL
DOUBLE PRECISION A(NDIM), B(NDIM), FUNCTN, EPS, ACC, WRKSTR
  1 (LENWRK), FINVAL
EXTERNAL FUNCTN
```

3. Description

The routine returns an estimate of a multi-dimensional integral over a hyper-rectangle (i.e., with constant limits), and also an estimate of the relative error. The user sets the relative accuracy required, supplies the integrand as a function subprogram (FUNCTN), and also sets the minimum and maximum acceptable number of calls to FUNCTN (in MINPTS and MAXPTS).

The routine operates by repeated subdivision of the hyper-rectangular region into smaller hyper-rectangles. In each subregion, the integral is estimated using a seventh-degree rule, and an error estimate is obtained by comparison with a fifth-degree rule which uses a subset of the same points. The fourth differences of the integrand along each co-ordinate axis are evaluated, and the subregion is marked for possible future subdivision in half along that co-ordinate axis which has the largest absolute fourth difference.

If the estimated errors, totalled over the subregions, exceed the requested relative error (or if fewer than MINPTS calls to FUNCTN have been made), further subdivision is necessary, and is
performed on the subregion with the largest estimated error, that subregion being halved along the appropriate co-ordinate axis.

The routine will fail if the requested relative error level has not been attained by the time MAXPTS calls to FUNCTN have been made; or, if the amount LENWRK of working storage is insufficient. A formula for the recommended value of LENWRK is given in Section 5. If a smaller value is used, and is exhausted in the course of execution, the routine switches to a less efficient mode of operation; only if this mode also breaks down is insufficient storage reported.

D01FCF is based on the HALF subroutine developed by van Dooren and de Ridder [1]. It uses a different basic rule, described by Genz and Malik [2].

4. References


5. Parameters

1: NDIM -- INTEGER Input
   On entry: the number of dimensions of the integral, n. Constraint: 2 <= NDIM <= 15.

2: A(NDIM) -- DOUBLE PRECISION array Input
   On entry: the lower limits of integration, ai , for i = 1,2,...,n.

3: B(NDIM) -- DOUBLE PRECISION array Input
   On entry: the upper limits of integration, bi , for i = 1,2,...,n.

4: MINPTS -- INTEGER Input/Output
   On entry: MINPTS must be set to the minimum number of integrand evaluations to be allowed. On exit: MINPTS contains the actual number of integrand evaluations used by D01FCF.

5: MAXPTS -- INTEGER Input
   On entry: the maximum number of integrand evaluations to be
allowed.
Constraints:

\[ \text{MAXPTS} \geq \text{MINPTS} \]

\[ \text{MAXPTS} \geq (\alpha), \]

\[ \text{NDIM} \quad 2 \]

where \( (\alpha) = 2 + 2 \times \text{NDIM} + 2 \times \text{NDIM} + 1 \).

6: \text{FUNCTN} -- \text{DOUBLE PRECISION FUNCTION}, supplied by the user.

External Procedure

\text{FUNCTN} must return the value of the integrand \( f \) at a given point.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION FUNCTION (NDIM,Z)}
\]

\[
\text{INTEGER NDIM}
\]

\[
\text{DOUBLE PRECISION Z(NDIM)}
\]

1: \text{NDIM} -- \text{INTEGER} \hspace{1cm} \text{Input}

On entry: the number of dimensions of the integral, \( n \).

2: \text{Z(NDIM)} -- \text{DOUBLE PRECISION array} \hspace{1cm} \text{Input}

On entry: the co-ordinates of the point at which the integrand must be evaluated.

\text{FUNCTN} must be declared as \text{EXTERNAL} in the (sub)program from which \text{D01FCF} is called. Parameters denoted as \text{Input} must not be changed by this procedure.

7: \text{EPS} -- \text{DOUBLE PRECISION} \hspace{1cm} \text{Input}

On entry: the relative error acceptable to the user. When the solution is zero or very small relative accuracy may not be achievable but the user may still set \text{EPS} to a reasonable value and check for the error exit \text{IFAIL} = 2. Constraint: \text{EPS} > 0.0.

8: \text{ACC} -- \text{DOUBLE PRECISION} \hspace{1cm} \text{Output}

On exit: the estimated relative error in \text{FINVAL}.

9: \text{LENWRK} -- \text{INTEGER} \hspace{1cm} \text{Input}

On entry: the dimension of the array \text{WRKSTR} as declared in the (sub)program from which \text{D01FCF} is called. Suggested value: for maximum efficiency, \text{LENWRK} \geq (\text{NDIM}+2)\times(1+\text{MAXPTS}/(\alpha)) (see parameter \text{MAXPTS} for \( (\alpha) \)).

If \text{LENWRK} is less than this, the routine will usually run
less efficiently and may fail. Constraint: LENWRK>=2*NDIM+4.

10: WRKSTR(LENWRK) -- DOUBLE PRECISION array Workspace
11: FINVAL -- DOUBLE PRECISION Output
On exit: the best estimate obtained for the integral.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).
For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
the output of an error message when soft failure occurs, set
IFAIL to 1.

6. Error Indicators and Warnings
Errors or warnings specified by the routine:

IFAIL= 1
On entry NDIM < 2,
or NDIM > 15,
or MAXPTS is too small,
or LENWRK<2*NDIM+4,
or EPS <= 0.0.

IFAIL= 2
MAXPTS was too small to obtain the required relative
accuracy EPS. On soft failure, FINVAL and ACC contain
estimates of the integral and the relative error, but ACC
will be greater than EPS.

IFAIL= 3
LENWRK was too small. On soft failure, FINVAL and ACC
contain estimates of the integral and the relative error,
but ACC will be greater than EPS.

7. Accuracy
A relative error estimate is output through the parameter ACC.
8. Further Comments

Execution time will usually be dominated by the time taken to evaluate the integrand FUNCTN, and hence the maximum time that could be taken will be proportional to MAXPTS.

9. Example

This example program estimates the integral

\[ \int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{4z \exp(2z \ z)}{1 + 3z \ + z^3} \ dz \ dz \ dz \ dz = 0.575364. \]

The accuracy requested is one part in 10,000.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D01GAF(3NAG)  Foundation Library (12/10/92)  D01GAF(3NAG)

D01 -- Quadrature
D01GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01GAF integrates a function which is specified numerically at four or more points, over the whole of its specified range, using third-order finite-difference formulae with error estimates, according to a method due to Gill and Miller.

2. Specification

SUBROUTINE D01GAF (X, Y, N, ANS, ER, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), Y(N), ANS, ER
3. Description

This routine evaluates the definite integral

\[ \int_{x_1}^{x_n} y(x) \, dx, \]

where the function \( y \) is specified at the \( n \)-points \( x_1, x_2, \ldots, x_n \), which should be all distinct, and in either ascending or descending order. The integral between successive points is calculated by a four-point finite-difference formula centred on the interval concerned, except in the case of the first and last intervals, where four-point forward and backward difference formulae respectively are employed. If \( n \) is less than 4, the routine fails. An approximation to the truncation error is integrated and added to the result. It is also returned separately to give an estimate of the uncertainty in the result. The method is due to Gill and Miller.

4. References


5. Parameters

1: \( X(N) \) -- DOUBLE PRECISION array
   Input
   On entry: the values of the independent variable, i.e., the \( x_1, x_2, \ldots, x_n \). Constraint: either \( X(1) < X(2) < \ldots < X(N) \) or \( X(1) > X(2) > \ldots > X(N) \).

2: \( Y(N) \) -- DOUBLE PRECISION array
   Input
   On entry: the values of the dependent variable \( y \) at the points \( x_i \), for \( i=1,2,\ldots,n \).

3: \( N \) -- INTEGER
   Input
   On entry: the number of points, \( n \). Constraint: \( N \geq 4 \).

4: \( ANS \) -- DOUBLE PRECISION
   Output
   On exit: the estimate of the integral.
5: ER -- DOUBLE PRECISION  
   Output  
   On exit: an estimate of the uncertainty in ANS.

6: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1  
   Indicates that fewer than four-points have been supplied to  
   the routine.

IFAIL= 2  
   Values of X are neither strictly increasing nor strictly  
   decreasing.

IFAIL= 3  
   Two points have the same X-value.

No error is reported arising from the relative magnitudes of ANS  
and ER on return, due to the difficulty when the true answer is  
zero.

7. Accuracy

No accuracy level is specified by the user before calling the  
routine but on return ABS(ER) is an approximation to, but not  
necessarily a bound for, |I-ANS|. If on exit IFAIL > 0, both ANS  
and ER are returned as zero.

8. Further Comments

The time taken by the routine depends on the number of points  
supplied, n.

In their paper, Gill and Miller [1] do not add the quantity ER to  
ANS before return. However, extensive tests have shown that a  
dramatic reduction in the error often results from such addition.  
In other cases, it does not make an improvement, but these tend  
to be cases of low accuracy in which the modified answer is not  
significantly inferior to the unmodified one. The user has the  
option of recovering the Gill-Miller answer by subtracting ER  
from ANS on return from the routine.
9. Example

The example program evaluates the integral

\[
\int_{0}^{2} \frac{1}{4(1+x)} \, dx = \pi
\]

reading in the function values at 21 unequally-spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Upon entry, unless LENWRK has been set to the minimum value 10*NDIM, the routine subdivides the integration region into a number of equal volume subregions. Inside each subregion the integral and the variance are estimated by means of pseudo-random sampling. All contributions are added together to produce an estimate for the whole integral and total variance. The variance along each co-ordinate axis is determined and the routine uses this information to increase the density and change the widths of the sub-intervals along each axis, so as to reduce the total variance. The total number of subregions is then increased by a factor of two and the program recycles for another iteration. The program stops when a desired accuracy has been reached or too many integral evaluations are needed for the next cycle.

4. References


5. Parameters

1: NDIM -- INTEGER Input
On entry: the number of dimensions of the integral, n.
Constraint: NDIM >= 1.

2: A(NDIM) -- DOUBLE PRECISION array Input
On entry: the lower limits of integration, $a_i$ for $i = 1, 2, \ldots, n$.

3: B(NDIM) -- DOUBLE PRECISION array Input
On entry: the upper limits of integration, $b_i$ for $i = 1, 2, \ldots, n$.

4: MINCLS -- INTEGER Input/Output
On entry: MINCLS must be set:

either to the minimum number of integrand evaluations to be allowed, in which case MINCLS >= 0;
or to a negative value. In this case the routine assumes that a previous call had been made with the same parameters NDIM, A and B and with either the same integrand (in which case D01GBF continues calculation) or a similar integrand (in which case D01GBF begins the calculation with the subdivision used in the last iteration of the previous call). See also WRKSTR. On exit: MINCLS contains the number of integrand evaluations actually used by D01GBF.

5: MAXCLS -- INTEGER

On entry: the maximum number of integrand evaluations to be allowed. In the continuation case this is the number of new integrand evaluations to be allowed. These counts do not include zero integrand values.
 Constraints:

\[
\text{MAXCLS} > \text{MINCLS}, \\
\text{MAXCLS} \geq 4 \times (\text{NDIM}+1).
\]

6: FUNCTN -- DOUBLE PRECISION FUNCTION, supplied by the user.

External Procedure

FUNCTN must return the value of the integrand \( f \) at a given point.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION FUNCTN (NDIM, X)} \\
\text{INTEGER NDIM} \\
\text{DOUBLE PRECISION X(NDIM)}
\]

1: NDIM -- INTEGER

On entry: the number of dimensions of the integral, \( n \).

2: X(NDIM) -- DOUBLE PRECISION array

On entry: the co-ordinates of the point at which the integrand must be evaluated.

FUNCTN must be declared as EXTERNAL in the (sub)program from which D01GBF is called. Parameters denoted as Input must not be changed by this procedure.

7: EPS -- DOUBLE PRECISION

On entry: the relative accuracy required. Constraint: EPS \( \geq 0.0 \).

8: ACC -- DOUBLE PRECISION

Output

On exit: the estimated relative accuracy of FINEST.

9: LENWRK -- INTEGER

On entry:
the dimension of the array WRKSTR as declared in the
(sub)program from which D01GBF is called.
For maximum efficiency, LENWRK should be about
\[ \frac{1}{NDIM} \times 3 \times NDIM \times (MAXCLS/4) + 7 \times NDIM. \]
If LENWRK is given the value 10*NDIM then the subroutine
uses only one iteration of a crude Monte Carlo method with
MAXCLS sample points. Constraint: LENWRK \geq 10*NDIM.

10: **WRKSTR(LENWRK)** -- DOUBLE PRECISION array
   Input/Output
   On entry: if MINCLS<0, WRKSTR must be unchanged from the
   previous call of D01GBF - except that for a new integrand
   WRKSTR(LENWRK) must be set to 0.0. See also MINCLS. On
   exit: WRKSTR contains information about the current sub-
   interval structure which could be used in later calls of
   D01GBF. In particular, WRKSTR(j) gives the number of sub-
   intervals used along the jth co-ordinate axis.

11: **FINEST** -- DOUBLE PRECISION
    Output
    On exit: the best estimate obtained for the integral.

12: **IFAIL** -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).
    For this routine, because the values of output parameters
    may be useful even if IFAIL /= 0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit. To suppress
    the output of an error message when soft failure occurs, set
    IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

**IFAIL**= 1
   On entry NDIM < 1,
   or \( \text{MINCLS} \geq \text{MAXCLS}, \)
   or \( \text{LENWRK} < 10\times\text{NDIM}, \)
   or \( \text{MAXCLS} < 4\times(\text{NDIM}+1), \)
   or \( \text{EPS} < 0.0. \)
CHAPTER 15.  CHAPTER N

IFAIL = 2
MAXCLS was too small for D01GBF to obtain the required relative accuracy EPS. In this case D01GBF returns a value of FINEST with estimated relative error ACC, but ACC will be greater than EPS. This error exit may be taken before MAXCLS non-zero integrand evaluations have actually occurred, if the routine calculates that the current estimates could not be improved before MAXCLS was exceeded.

7. Accuracy

A relative error estimate is output through the parameter ACC. The confidence factor is set so that the actual error should be less than ACC 90% of the time. If a user desires a higher confidence level then a smaller value of EPS should be used.

8. Further Comments

The running time for D01GBF will usually be dominated by the time used to evaluate the integrand FUNCTN, so the maximum time that could be used is approximately proportional to MAXCLS.

For some integrands, particularly those that are poorly behaved in a small part of the integration region, D01GBF may terminate with a value of ACC which is significantly smaller than the actual relative error. This should be suspected if the returned value of MINCLS is small relative to the expected difficulty of the integral. Where this occurs, D01GBF should be called again, but with a higher entry value of MINCLS (e.g. twice the returned value) and the results compared with those from the previous call.

The exact values of FINEST and ACC on return will depend (within statistical limits) on the sequence of random numbers generated within D01GBF by calls to G05CAF. Separate runs will produce identical answers unless the part of the program executed prior to calling D01GBF also calls (directly or indirectly) routines from Chapter G05, and the series of such calls differs between runs. If desired, the user may ensure the identity or difference between runs of the results returned by D01GBF, by calling G05CBF or G05CCF respectively, immediately before calling D01GBF.

9. Example

This example program calculates the integral

\[\int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} 4x \exp(2x \ x ) \, dx \, dx \, dx = 0.575364.\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F)) -> Result
++ d01ajf(a,b,epsabs,epsrel,\(lw,liw,ifail,f\))
++ is a general-purpose integrator which calculates an
++ approximation to the integral of a function \(f(x)\) over a finite
++ interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01ajf}.

d01akf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
++ d01akf(a,b,epsabs,epsrel,\(lw,liw,ifail,f\))
++ is an adaptive integrator, especially suited to
++ oscillating, non-singular integrands, which calculates an
++ approximation to the integral of a function \(f(x)\) over a finite
++ interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01akf}.

d01alf : (DoubleFloat,DoubleFloat,Integer,Matrix DoubleFloat,_
DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
++ d01alf(a,b,npts,points,epsabs,epsrel,\(lw,liw,ifail,f\))
++ is a general purpose integrator which calculates an
++ approximation to the integral of a function \(f(x)\) over a finite
++ interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01alf}.

d01amf : (DoubleFloat,Integer,DoubleFloat,DoubleFloat,_
Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
++ d01amf(bound,inf,epsabs,epsrel,\(lw,liw,ifail,f\))
++ calculates an approximation to the integral of a function
++ \(f(x)\) over an infinite or semi-infinite interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01amf}.

d01anf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,_
DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(G))) -> Result
++ d01anf(a,b,omega,key,epsabs,epsrel,\(lw,liw,ifail,g\))
++ calculates an approximation to the sine or the cosine
++ transform of a function \(g(x)\) over \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01anf}.

d01apf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(G))) -> Result
++ d01apf(a,b,alfa,\(\alpha,\beta\),key,epsabs,epsrel,\(lw,liw,ifail,g\))
++ is an adaptive integrator which calculates an
++ approximation to the integral of a function \(g(x)w(x)\) over a
++ finite interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01apf}.

d01aqf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(G))) -> Result
++ d01aqf(a,b,c,epsabs,epsrel,\(lw,liw,ifail,g\))
++ calculates an approximation to the Hilbert transform of a
++ function \(g(x)\) over \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01aqf}. 
PACKAGE NAGD01 NAGINTEGRATIONPACKAGE

d01asf : (DoubleFloat,DoubleFloat,Integer,DoubleFloat,_
    Integer,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp1(G))) -> Result
++ d01asf(a,omega,key,epsabs,limlst,lw,liw,ifail,g)
++ calculates an approximation to the sine or the cosine
++ transform of a function g over [a,infty):
++ See \downlink{Manual Page}{manpageXXd01asf}.

d01bbf : (DoubleFloat,DoubleFloat,Integer,Integer,_
    Integer,Integer,Integer) -> Result
++ d01bbf(a,b,itype,n,gtype,ifail)
++ returns the weight appropriate to a
++ Gaussian quadrature.
++ The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-
++ Laguerre and Gauss-Hermite.
++ See \downlink{Manual Page}{manpageXXd01bbf}.

d01fcf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    DoubleFloat, Integer,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp4(FUNCTN))) -> Result
++ d01fcf(ndim,a,b,maxpts,eps,lenwrk,minpts,iftail,functn)
++ attempts to evaluate a multi-dimensional integral (up to
++ 15 dimensions), with constant and finite limits, to a specified
++ relative accuracy, using an adaptive subdivision strategy.
++ See \downlink{Manual Page}{manpageXXd01fcf}.

d01gaf : (Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer) -> Result
++ d01gaf(x,y,n,ifail)
++ integrates a function which is specified numerically at
++ four or more points, over the whole of its specified range, using
++ third-order finite-difference formulae with error estimates,
++ according to a method due to Gill and Miller.
++ See \downlink{Manual Page}{manpageXXd01gaf}.

d01gbf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    DoubleFloat, Integer,Integer,Integer,Integer,Integer,
    Matrix DoubleFloat,Integer,Integer,_
    Union(fn:FileName,fp:Asp4(FUNCTN))) -> Result
++ d01gbf(ndim,a,b,maxcls,eps,lenwrk,mincls,wrkstr,ifail,functn)
++ returns an approximation to the integral of a function
++ over a hyper-rectangular region, using a Monte Carlo method. An
++ approximate relative error estimate is also returned. This
++ routine is suitable for low accuracy work.
++ See \downlink{Manual Page}{manpageXXd01gbf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName, fp: Asp1(F))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)

d01ajf(aArg: DoubleFloat, bArg: DoubleFloat, epsAbsArg: DoubleFloat, _
    epsRelArg: Integer, lwArg: Integer, liwArg: Integer, _
    ifailArg: Integer, fArg: Union(fn: FileName, fp: Asp1(F))): Result ==
    pushFortranOutputStack(fFilename := aspFilename "f")$FOP
    if fArg case fn
        then outputAsFortran(fArg.fn)
        else outputAsFortran(fArg.fp)
    popFortranOutputStack()$FOP
    ![invokeNagman([[fFilename]]$Lisp, _
        "d01ajf", _
        "w": S,"iw": S]$Lisp, _
        "ifail": S]$Lisp$]
    ]$Lisp_,
    ![invokeNagman([[fFilename]]$Lisp, _
        "d01ajf", _
        "w": S,"iw": S]$Lisp, _
        "ifail": S]$Lisp$]
    ]$Lisp_,

pretend List (Record(key: Symbol, entry: Any))$Result

d01akf(aArg: DoubleFloat, bArg: DoubleFloat, epsAbsArg: DoubleFloat, _
    epsRelArg: DoubleFloat, lwArg: Integer, liwArg: Integer, _
    ifailArg: Integer, fArg: Union(fn: FileName, fp: Asp1(F))): Result ==
    pushFortranOutputStack(fFilename := aspFilename "f")$FOP
    if fArg case fn
        then outputAsFortran(fArg.fn)
        else outputAsFortran(fArg.fp)
    popFortranOutputStack()$FOP
    ![invokeNagman([[fFilename]]$Lisp, _
        "d01akf", _
        "w": S,"iw": S]$Lisp, _
        "ifail": S]$Lisp$]
    ]$Lisp,
d01alf(aArg:DoubleFloat,bArg:DoubleFloat,nptsArg:Integer,_
pointsArg:Matrix DoubleFloat,epsabsArg:DoubleFloat,_
epsrelArg:DoubleFloat,_
lwArg:Integer,liwArg:Integer,ifailArg:Integer,_
fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
pushFortranOutputStack(fFilename := aspFilename "f")$FOP
if fArg case fn
   then outputAsFortran(fArg.fn)
   else outputAsFortran(fArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fFilename]$Lisp,_
"d01alf",_
  "f"::S,"points"::S,"w"::S,"iw"::S]$Lisp,_
  "epsabs"::S,"epsrel"::S,"result"::S,"abserr"::S_,
  "w"::S,"lw"::S]$Lisp,"f"::S]$Lisp_]
  "ifail"::S]$Lisp_]
]$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

d01amf(boundArg:DoubleFloat,infArg:Integer,epsabsArg:DoubleFloat,_
epsrelArg:DoubleFloat,_
lwArg:Integer,liwArg:Integer,ifailArg:Integer,fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
pushFortranOutputStack(fFilename := aspFilename "f")$FOP
if fArg case fn
   then outputAsFortran(fArg.fn)
   else outputAsFortran(fArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fFilename]$Lisp,_
"d01amf",_
  "w"::S,"iw"::S]$Lisp,_
pretend List (Record(key:Symbol,entry:Any))$Result
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename]$Lisp, _
  "d01anf", _
  ["a": S, "b": S, "omega": S, "key": S, "epsabs": S, _
   "epsrel": S, "lw": S, "liw": S, "result": S, "abserr": S, _
   "ifail": S, "g": S, "w": S, "iw": S]$Lisp, _
  ["result": S, "abserr": S, "w": S, "iw": S, "ifail": S]$Lisp, _
  [([boundArg::Any, infArg::Any, epsabsArg::Any, epsrelArg::Any, _
     lwArg::Any, liwArg::Any, ifailArg::Any ])
  @List Any]$Lisp)$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result

invokeNagman([gFilename]$Lisp, _
  "d01anf", _
  ["a": S, "b": S, "omega": S, "key": S, "epsabs": S, _
   "epsrel": S, "lw": S, "liw": S, "result": S, "abserr": S, _
   "ifail": S, "g": S, "w": S, "iw": S]$Lisp, _
  ["result": S, "abserr": S, "w": S, "iw": S, "ifail": S]$Lisp, _
  [([boundArg::Any, infArg::Any, epsabsArg::Any, epsrelArg::Any, _
     lwArg::Any, liwArg::Any, ifailArg::Any ])
  @List Any]$Lisp)$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result

pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename]$Lisp, _
  "d01apf", _
  ["a": S, "b": S, "alfa": S, "beta": S, "key": S, _
   "epsabs": S, "epsrel": S, "lw": S, "liw": S, "result": S, _
   "abserr": S, "ifail": S, "g": S, "w": S, "iw": S]$Lisp, _
   ["result": S, "abserr": S, "w": S, "iw": S, "ifail": S]$Lisp, _
   [([boundArg::Any, infArg::Any, epsabsArg::Any, epsrelArg::Any, _
     lwArg::Any, liwArg::Any, ifailArg::Any ])
  @List Any]$Lisp)$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result

invokeNagman([gFilename]$Lisp, _
  "d01apf", _
  ["a": S, "b": S, "alfa": S, "beta": S, "key": S, _
   "epsabs": S, "epsrel": S, "lw": S, "liw": S, "result": S, _
   "abserr": S, "ifail": S, "g": S, "w": S, "iw": S]$Lisp, _
   ["result": S, "abserr": S, "w": S, "iw": S, "ifail": S]$Lisp, _
   [([boundArg::Any, infArg::Any, epsabsArg::Any, epsrelArg::Any, _
     lwArg::Any, liwArg::Any, ifailArg::Any ])
  @List Any]$Lisp)$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result
Pretend List (Record(key:Symbol,entry:Any))$Result

d01aqf(aArg:DoubleFloat,bArg:DoubleFloat,cArg:DoubleFloat,epsabsArg:DoubleFloat,epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,ifailArg:Integer,gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then 
  else 
}

pretend List (Record(key:Symbol,entry:Any))$Result

pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then 
  else 
}
"d01asf", \
["a"::S,"omega"::S,"key"::S,"epsabs"::S,"limlst"::S_, \
]$Lisp,_ 
["result"::S,"abserr"::S,"lst"::S,"erlst"::S,"rslst"::S_, \
"ierlst"::S,"lw"::S,"w"::S,"g"::S]$Lisp,_ 
["double"::S,"a"::S,"omega"::S,"epsabs"::S_, \
,"result"::S,"abserr"::S,"limlst"::S]$Lisp,_ 
["erlst"::S,"rslst"::S]$Lisp,["w"::S,"lw"::S]$Lisp,"g"::S]$Lisp_ , 
["integer"::S,"key"::S,"limlst"::S,"lw"::S_, \
["iw"::S,"lw"::S]$Lisp,"ifail"::S]$Lisp_ ]$Lisp, 
[[(aArg::Any,omegaArg::Any,keyArg::Any,epsabsArg::Any_, \
    limlstArg::Any,lwArg::Any,liwArg::Any,ifailArg::Any ])_ 
@List Any]@List Any)@List Any)@List Any 
pretend List (Record(key:Symbol,entry:Any))$Result

d01bbf(aArg:DoubleFloat,bArg:DoubleFloat,itypeArg:Integer,_ 
    [(invokeNagman(NIL$Lisp, 
        "d01bbf", 
        ["a"::S,"b"::S,"itype"::S,"n"::S,"gtype"::S_ , 
        "ifail"::S,"weight"::S,"abscis"::S]$Lisp_, 
        ["weight"::S,"abscis"::S]$Lisp_, 
        ["double"::S,"a"::S,"b"::S,"weight"::S,"n"::S]$Lisp_ , 
        ["abscis"::S,"n"::S]$Lisp]$Lisp_ , 
        ["integer"::S,"itype"::S,"n"::S,"gtype"::S_ , 
        "ifail"::S]$Lisp_ ]$Lisp_, 
    [([aArg::Any,bArg::Any,itypeArg::Any,nArg::Any_, 
    gtypeArg::Any,ifailArg::Any ])_ 
    @List Any]@List Any)@List Any) 
pretend List (Record(key:Symbol,entry:Any))$Result

d01fcf(ndimArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat, 
    maxptsArg:Integer,epsArg:DoubleFloat,lenwrkArg:Integer, 
    minptsArg:Integer,ifailArg:Integer, 
    functnArg:Union(fn:FileName,fp:Asp4(FUNCTN))): Result == 
    pushFortranOutputStack(functnFilename := aspFilename "functn")$FOP 
    if functnArg case fn 
    then outputAsFortran(functnArg.fn) 
    else outputAsFortran(functnArg.fp) 
    popFortranOutputStack()$FOP 
    [(invokeNagman([functnFilename]$Lisp, 
        "d01fcf", 

d01gaf(xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,nArg:Integer,ifailArg:Integer): Result ==
pretend List (Record(key:Symbol,entry:Any))$Result

pushFortranOutputStack(functnFilename := aspFilename "functn")$FOP
if functnArg case fn then outputAsFortran(functnArg.fn)
else outputAsFortran(functnArg.fp)
popFortranOutputStack()$FOP
pretend List (Record(key:Symbol,entry:Any))$Result

package NAGE01 NagInterpolationPackage

— NagInterpolationPackage.input —

)set break resume
)sys rm -f NagInterpolationPackage.output
)spool NagInterpolationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagInterpolationPackage
--E 1

)spool
)lisp (bye)

— NagInterpolationPackage.help —

This package uses the NAG Library to calculate the interpolation of a function of one or two variables. When provided with the value of the function (and possibly one or more of its lowest-order
derivatives) at each of a number of values of the variable(s),
the routines provide either an interpolating function or an
interpolated value. For some of the interpolating functions,
there are supporting routines to evaluate, differentiate or
integrate them.

E01 -- Interpolation

Chapter E01
Interpolation

1. Scope of the Chapter

This chapter is concerned with the interpolation of a function of
one or two variables. When provided with the value of the
function (and possibly one or more of its lowest-order
derivatives) at each of a number of values of the variable(s),
the routines provide either an interpolating function or an
interpolated value. For some of the interpolating functions,
there are supporting routines to evaluate, differentiate or
integrate them.

2. Background to the Problems

In motivation and in some of its numerical processes, this
chapter has much in common with Chapter E02 (Curve and Surface
Fitting). For this reason, we shall adopt the same terminology
and refer to dependent variable and independent variable(s)
instead of function and variable(s). Where there is only one
independent variable, we shall denote it by \( x \) and the dependent
variable by \( y \). Thus, in the basic problem considered in this
chapter, we are given a set of distinct values \( x_1, x_2, \ldots, x_m \)
and a corresponding set of values \( y_1, y_2, \ldots, y_m \) of \( y \), and we shall
describe the problem as being one of interpolating the data
points \((x_r, y_r)\), rather than interpolating a function. In modern
usage, however, interpolation can have either of two rather
different meanings, both relevant to routines in this chapter.
They are

(a) the determination of a function of \( x \) which takes the value \( y_r \)
at \( x = x_r \), for \( r = 1, 2, \ldots, m \) (an interpolating function or
interpolant),

(b) the determination of the value (interpolated value or
interpolate) of an interpolating function at any given value,
say \( x \), of \( x \) within the range of the \( x \) (so as to estimate the \( r \) value at \( x \) of the function underlying the data).

The latter is the older meaning, associated particularly with the use of mathematical tables. The term 'function underlying the data', like the other terminology described above, is used so as to cover situations additional to those in which the data points have been computed from a known function, as with a mathematical table. In some contexts, the function may be unknown, perhaps representing the dependency of one physical variable on another, say temperature upon time.

Whether the underlying function is known or unknown, the object of interpolation will usually be to approximate it to acceptable accuracy by a function which is easy to evaluate anywhere in some range of interest. Piecewise polynomials such as cubic splines (see Section 2.2 of the E02 Chapter Introduction for definitions of terms in this case), being easy to evaluate and also capable of approximating a wide variety of functions, are the types of function mostly used in this chapter as interpolating functions.

Piecewise polynomials also, to a large extent, avoid the well-known problem of large unwanted fluctuations which can arise when interpolating a data set with a simple polynomial. Fluctuations can still arise but much less frequently and much less severely than with simple polynomials. Unwanted fluctuations are avoided altogether by a routine using piecewise cubic polynomials having only first derivative continuity. It is designed especially for monotonic data, but for other data still provides an interpolant which increases, or decreases, over the same intervals as the data.

The concept of interpolation can be generalised in a number of ways. For example, we may be required to estimate the value of the underlying function at a value \( x \) outside the range of the data. This is the process of extrapolation. In general, it is a good deal less accurate than interpolation and is to be avoided whenever possible.

Interpolation can also be extended to the case of two independent variables. We shall denote these by \( x \) and \( y \), and the dependent variable by \( f \). Methods used depend markedly on whether or not the data values of \( f \) are given at the intersections of a rectangular mesh in the \((x, y)\)-plane. If they are, bicubic splines (see Section 2.3.2 of the E02 Chapter Introduction) are very suitable and usually very effective for the problem. For other cases, perhaps where the \( f \) values are quite arbitrarily scattered in the
(x,y)-plane, polynomials and splines are not at all appropriate and special forms of interpolating function have to be employed. Many such forms have been devised and two of the most successful are in routines in this chapter. They both have continuity in first, but not higher, derivatives.

2.1. References


3. Recommendations on Choice and Use of Routines

3.1. General

Before undertaking interpolation, in other than the simplest cases, the user should seriously consider the alternative of using a routine from Chapter E02 to approximate the data by a polynomial or spline containing significantly fewer coefficients than the corresponding interpolating function. This approach is much less liable to produce unwanted fluctuations and so can often provide a better approximation to the function underlying the data.

When interpolation is employed to approximate either an underlying function or its values, the user will need to be satisfied that the accuracy of approximation achieved is adequate. There may be a means for doing this which is particular to the application, or the routine used may itself provide a means. In other cases, one possibility is to repeat the interpolation using one or more extra data points, if they are available, or otherwise one or more fewer, and to compare the results. Other possibilities, if it is an interpolating function which is determined, are to examine the function graphically, if that gives sufficient accuracy, or to observe the behaviour of the differences in a finite-difference table, formed from evaluations of the interpolating function at equally-spaced values of x over the range of interest. The spacing should be small enough to cause the typical size of the differences to decrease as the order of difference increases.

3.2. One Independent Variable

E01BAF computes an interpolating cubic spline, using a particular choice for the set of knots which has proved generally satisfactory in practice. If the user wishes to choose a different set, a cubic spline routine from Chapter E02, namely
E02BAF, may be used in its interpolating mode, setting NCAP7 = M+4 and all elements of the parameter W to unity. These routines provide the interpolating function in B-spline form (see Section 2.2.2 in the E02 Chapter Introduction). Routines for evaluating, differentiating and integrating this form are discussed in Section 3.7 of the E02 Chapter Introduction.

The cubic spline does not always avoid unwanted fluctuations, especially when the data show a steep slope close to a region of small slope, or when the data inadequately represent the underlying curve. In such cases, E01BEF can be very useful. It derives a piecewise cubic polynomial (with first derivative continuity) which, between any adjacent pair of data points, either increases all the way, or decreases all the way (or stays constant). It is especially suited to data which are monotonic over their whole range.

In this routine, the interpolating function is represented simply by its value and first derivative at the data points. Supporting routines compute its value and first derivative elsewhere, as well as its definite integral over an arbitrary interval.

3.3. Two Independent Variables

3.3.1. Data on a rectangular mesh

Given the value of the dependent variable f at the point \((x_q, y_r)\) in the plane of the independent variables x and y, for each \(q = 1, 2, ..., m\) and \(r = 1, 2, ..., n\) (so that the points \((x_q, y_r)\) lie at the \(m \times n\) intersections of a rectangular mesh), E01DAF computes an interpolating bicubic spline, using a particular choice for each of the spline's knot-set. This choice, the same as in E01BAF, has proved generally satisfactory in practice. If, instead, the user wishes to specify his own knots, a routine from Chapter E02, namely E02DAF, may be adapted (it is more cumbersome for the purpose, however, and much slower for larger problems). Using \(m\) and \(n\) in the above sense, the parameter \(M\) must be set to \(m \times n\), PX and PY must be set to \(m + 4\) and \(n + 4\) respectively and all elements of \(W\) should be set to unity. The recommended value for EPS is zero.

3.3.2. Arbitrary data

As remarked at the end of Section 2, special types of interpolating are required for this problem, which can often be difficult to solve satisfactorily. Two of the most successful are employed in E01SAF and E01SEF, the two routines which (with their
respective evaluation routines E01SBF and E01SFF) are provided for the problem. Definitions can be found in the routine documents. Both interpolants have first derivative continuity and are 'local', in that their value at any point depends only on data in the immediate neighbourhood of the point. This latter feature is necessary for large sets of data to avoid prohibitive computing time.

The relative merits of the two methods vary with the data and it is not possible to predict which will be the better in any particular case.

3.4. Index

Derivative, of interpolant from E01BEF E01BGF

Evaluation, of interpolant
  from E01BEF E01BFF
  from E01SAF E01SBF
  from E01SEF E01SFF

Extrapolation, one variable E01BEF

Integration (definite) of interpolant from E01BEF E01BHF

Interpolated values, one variable, from interpolant from E01BEF E01BFF

Interpolated values, two variables,
  from interpolant from E01SAF E01SBF
  from interpolant from E01SEF E01SFF

Interpolating function, one variable,
  cubic spline E01BAF
  other piecewise polynomial E01BEF

Interpolating function, two variables
  bicubic spline E01DAF
  other piecewise polynomial E01SAF
  modified Shepard method E01SEF

E01 -- Interpolation Contents -- E01
Chapter E01

Interpolation

E01BAF Interpolating functions, cubic spline interpolant, one variable

E01BEF Interpolating functions, monotonicity-preserving, piecewise cubic Hermite, one variable

E01BFF Interpolated values, interpolant computed by E01BEF, function only, one variable,
E01BGF  Interpolated values, interpolant computed by E01BEF, function and 1st derivative, one variable
E01BHF  Interpolated values, interpolant computed by E01BEF, definite integral, one variable
E01DAF  Interpolating functions, fitting bicubic spline, data on rectangular grid
E01SAF  Interpolating functions, method of Renka and Cline, two variables
E01SBF  Interpolated values, evaluate interpolant computed by E01SAF, two variables
E01SEF  Interpolating functions, modified Shepard’s method, two variables
E01SFF  Interpolated values, evaluate interpolant computed by E01SEF, two variables

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E01 -- Interpolation
E01BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01BAF determines a cubic spline interpolant to a given set of data.

2. Specification

SUBROUTINE E01BAF (M, X, Y, LAMDA, C, LCK, WRK, LWRK, IFAIL)
INTEGER M, LCK, LWRK, IFAIL
DOUBLE PRECISION X(M), Y(M), LAMDA(LCK), C(LCK), WRK(LWRK)

3. Description

This routine determines a cubic spline $s(x)$, defined in the range $x \leq x \leq x$, which interpolates (passes exactly through) the set of $m$ data points $(x_i, y_i)$, for $i=1,2,...,m$, where $m \geq 4$ and $x < x < ... < x$
end conditions are not imposed. The spline interpolant chosen has $m-4$ interior knots $(\lambda_5, \lambda_6, \ldots, \lambda_m)$, which are set to the values of $x_3, x_4, \ldots, x_{m-2}$ respectively. This spline is represented in its B-spline form (see Cox [1]):

$$s(x) = \sum_{i=1}^{m} c_i N_i(x),$$

where $N_i(x)$ denotes the normalised B-Spline of degree 3, defined upon the knots $(\lambda_1, \lambda_2, \ldots, \lambda_m)$, and $c_{i}$ denotes its coefficient, whose value is to be determined by the routine.

The use of B-splines requires eight additional knots $(\lambda_1, 2, 3, 4, \lambda_{m+1}, \lambda_{m+2}, \lambda_{m+3}, \lambda_{m+4})$ to be specified; the routine sets the first four of these to $x_1$ and the last four to $x_m$.

The algorithm for determining the coefficients is as described in [1] except that QR factorization is used instead of LU decomposition. The implementation of the algorithm involves setting up appropriate information for the related routine E02BAF followed by a call of that routine. (For further details of E02BAF, see the routine document.)

Values of the spline interpolant, or of its derivatives or definite integral, can subsequently be computed as detailed in Section 8.

4. References


5. Parameters

1: M -- INTEGER
   Input
   On entry: m, the number of data points. Constraint: M \geq 4.

2: X(M) -- DOUBLE PRECISION array
   Input
   On entry: X(i) must be set to x , the ith data value of the
   independent variable x, for i=1,2,...,m. Constraint: X(i) <
   X(i+1), for i=1,2,...,M-1.

3: Y(M) -- DOUBLE PRECISION array
   Input
   On entry: Y(i) must be set to y , the ith data value of the
   dependent variable y, for i=1,2,...,m.

4: LAMDA(LCK) -- DOUBLE PRECISION array
   Output
   On exit: the value of (lambda) , the ith knot, for
   i=1,2,...,m+4.

5: C(LCK) -- DOUBLE PRECISION array
   Output
   On exit: the coefficient c of the B-spline N (x), for
   i=1,2,...,m. The remaining elements of the array are not
   used.

6: LCK -- INTEGER
   Input
   On entry:
   the dimension of the arrays LAMDA and C as declared in the
   (sub)program from which E01BAF is called.
   Constraint: LCK \geq M + 4.

7: WRK(LWRK) -- DOUBLE PRECISION array
   Workspace

8: LWRK -- INTEGER
   Input
   On entry:
   the dimension of the array WRK as declared in the
   (sub)program from which E01BAF is called.
   Constraint: LWRK \geq 6*M+16.

9: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
On entry $M < 4$,
or $LCK < M+4$,
or $LWRK < 6*M+16$.

IFAIL= 2
The $X$-values fail to satisfy the condition

$$X(1) < X(2) < X(3) < \ldots < X(M).$$

7. Accuracy

The rounding errors incurred are such that the computed spline is
an exact interpolant for a slightly perturbed set of ordinates $y + \delta y$. The ratio of the root-mean-square value of the

$$\delta y_i$$

of the relative machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to $M$.

All the $X_i$ are used as knot positions except $X_1$ and $X_2$. This
choice of knots (see Cox [2]) means that $s(x)$ is composed of $m-3$
cubic arcs as follows. If $m=4$, there is just a single arc space
spanning the whole interval $x_1$ to $x_4$. If $m>4$, the first and last
arcs span the intervals $x_1$ to $x_4$ and $x_3$ to $x_m$ respectively.
Additionally if $m>6$, the $i$th arc, for $i=2,3,\ldots,m-4$ spans the
interval $x_i$ to $x_{i+2}$.

After the call

CALL E01BAF $(M, X, Y, LAMDA, C, LCK, WRK, LWRK, IFAIL)$

the following operations may be carried out on the interpolant
$s(x)$.

The value of $s(x)$ at $x = XVAL$ can be provided in the real
variable $SVAL$ by the call
The values of $s(x)$ and its first three derivatives at $x = XVAL$ can be provided in the real array SDIF of dimension 4, by the call:

```
CALL E02BBF (M+4, LAMDA, C, XVAL, SVAL, IFAIL)
```

Here LEFT must specify whether the left- or right-hand value of the third derivative is required (see E02BCF for details).

The value of the integral of $s(x)$ over the range $x_1$ to $x_m$ provided in the real variable SINT by:

```
CALL E02BDF (M+4, LAMDA, C, SINT, IFAIL)
```

9. Example

The example program sets up data from 7 values of the exponential function in the interval 0 to 1. E01BAF is then called to compute a spline interpolant to these data.

The spline is evaluated by E02BBF, at the data points and at points halfway between each adjacent pair of data points, and the spline values and the values of $e$ are printed out.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
SUBROUTINE E01BEF (N, X, F, D, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), F(N), D(N)

3. Description

This routine estimates first derivatives at the set of data points \((x_r, f_r)\), for \(r=1,2,...,n\), which determine a piecewise cubic Hermite interpolant to the data, that preserves monotonicity over ranges where the data points are monotonic. If the data points are only piecewise monotonic, the interpolant will have an extremum at each point where monotonicity switches direction. The estimates of the derivatives are computed by a formula due to Brodlie, which is described in Fritsch and Butland [1], with suitable changes at the boundary points.

The routine is derived from routine PCHIM in Fritsch [2].

Values of the computed interpolant, and of its first derivative and definite integral, can subsequently be computed by calling E01BFF, E01BGF and E01BHF, as described in Section 8.

4. References


5. Parameters

1: N -- INTEGER Input
   On entry: n, the number of data points. Constraint: \(N \geq 2\).

2: X(N) -- DOUBLE PRECISION array Input
   On entry: \(X(r)\) must be set to \(x_r\), the \(r\)th value of the \(r\)th independent variable (abscissa), for \(r=1,2,...,n\).
   Constraint: \(X(r) < X(r+1)\).

3: F(N) -- DOUBLE PRECISION array Input
   On entry: \(F(r)\) must be set to \(f_r\), the \(r\)th value of the \(r\)th dependent variable (ordinate), for \(r=1,2,...,n\).

4: D(N) -- DOUBLE PRECISION array Output
   On exit: estimates of derivatives at the data points. \(D(r)\)
contains the derivative at X(r).

5: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

   Errors detected by the routine:

   If on entry IFAIL = 0 or -1, explanatory error messages are
   output on the current error message unit (as defined by X04AAF).

   IFAIL = 1
   On entry N < 2.

   IFAIL = 2
   The values of X(r), for r = 1, 2, ..., N, are not in strictly
   increasing order.

7. Accuracy

   The computational errors in the array D should be negligible in
   most practical situations.

8. Further Comments

   The time taken by the routine is approximately proportional to n.

   The values of the computed interpolant at the points PX(i), for
   i = 1, 2, ..., M, may be obtained in the real array PF, of length at
   least M, by the call:

   CALL E01BFF(N,X,F,D,M,PX,PF,IFAIL)

   where N, X and F are the input parameters to E01BEF and D is the
   output parameter from E01BEF.

   The values of the computed interpolant at the points PX(i), for i
   = 1, 2, ..., M, together with its first derivatives, may be obtained
   in the real arrays PF and PD, both of length at least M, by the
   call:

   CALL E01BGF(N,X,F,D,M,PX,PF,PD,IFAIL)
where \( N, X, F \) and \( D \) are as described above.

The value of the definite integral of the interpolant over the interval \( A \) to \( B \) can be obtained in the real variable \( PINT \) by the call:

\[
\text{CALL E01BHF}(N, X, F, D, A, B, PINT, IFAIL)
\]

where \( N, X, F \) and \( D \) are as described above.

9. Example

This example program reads in a set of data points, calls E01BEF to compute a piecewise monotonic interpolant, and then calls E01BFF to evaluate the interpolant at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
4. References


5. Parameters

1: \( N \) -- INTEGER  
   Input

2: \( X(N) \) -- DOUBLE PRECISION array  
   Input

3: \( F(N) \) -- DOUBLE PRECISION array  
   Input

4: \( D(N) \) -- DOUBLE PRECISION array  
   Input
   On entry: \( N, X, F \) and \( D \) must be unchanged from the previous call of E01BEF.

5: \( M \) -- INTEGER  
   Input
   On entry: \( m \), the number of points at which the interpolant is to be evaluated. Constraint: \( M \geq 1 \).

6: \( PX(M) \) -- DOUBLE PRECISION array  
   Input
   On entry: the \( m \) values of \( x \) at which the interpolant is to be evaluated.

7: \( PF(M) \) -- DOUBLE PRECISION array  
   Output
   On exit: \( PF(i) \) contains the value of the interpolant evaluated at the point \( PX(i) \), for \( i=1,2,...,m \).

8: \( IFAIL \) -- INTEGER  
   Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \( IFAIL = 0 \) or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

\( IFAIL=1 \)
   On entry \( N < 2 \).

\( IFAIL=2 \)
   The values of \( X(r) \), for \( r = 1,2,...,N \), are not in strictly
increasing order.

IFAIL= 3
On entry M < 1.

IFAIL= 4
At least one of the points PX(i), for i = 1,2,...,M, lies outside the interval [X(1),X(N)], and extrapolation was performed at all such points. Values computed at such points may be very unreliable.

7. Accuracy
The computational errors in the array PF should be negligible in most practical situations.

8. Further Comments
The time taken by the routine is approximately proportional to the number of evaluation points, m. The evaluation will be most efficient if the elements of PX are in non-decreasing order (or, more generally, if they are grouped in increasing order of the intervals [X(r-1),X(r)]). A single call of E01BFF with m>1 is more efficient than several calls with m=1.

9. Example
This example program reads in values of N, X, F and D, and then calls E01BFF to evaluate the interpolant at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E01 -- Interpolation
E01BGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
E01BGF evaluates a piecewise cubic Hermite interpolant and its first derivative at a set of points.
2. Specification

```fortran
SUBROUTINE E01BGF (N, X, F, D, M, PX, PF, PD, IFAIL)
  INTEGER N, M, IFAIL
  DOUBLE PRECISION X(N), F(N), D(N), PX(M), PF(M), PD(M)
```

3. Description

This routine evaluates a piecewise cubic Hermite interpolant, as computed by E01BEF, at the points PX(i), for i=1,2,...,m. The first derivatives at the points are also computed. If any point lies outside the interval from X(1) to X(N), values of the interpolant and its derivative are extrapolated from the nearest extreme cubic, and a warning is returned.

If values of the interpolant only, and not of its derivative, are required, E01BFF should be used.

The routine is derived from routine PCHIP in Fritsch [1].

4. References


5. Parameters

1: N -- INTEGER  
Input

2: X(N) -- DOUBLE PRECISION array  
Input

3: F(N) -- DOUBLE PRECISION array  
Input

4: D(N) -- DOUBLE PRECISION array  
Input
On entry: N, X, F and D must be unchanged from the previous call of E01BEF.

5: M -- INTEGER  
Input
On entry: m, the number of points at which the interpolant is to be evaluated. Constraint: M >= 1.

6: PX(M) -- DOUBLE PRECISION array  
Input
On entry: the m values of x at which the interpolant is to be evaluated.

7: PF(M) -- DOUBLE PRECISION array  
Output
On exit: PF(i) contains the value of the interpolant evaluated at the point PX(i), for i=1,2,...,m.

8: PD(M) -- DOUBLE PRECISION array  
Output
On exit: PD(i) contains the first derivative of the interpolant evaluated at the point PX(i), for i=1,2,...,m.

9: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry N < 2.

IFAIL= 2
The values of X(r), for r = 1,2,...,N, are not in strictly increasing order.

IFAIL= 3
On entry M < 1.

IFAIL= 4
At least one of the points PX(i), for i = 1,2,...,M, lies outside the interval [X(1),X(N)], and extrapolation was performed at all such points. Values computed at these points may be very unreliable.

7. Accuracy
The computational errors in the arrays PF and PD should be negligible in most practical situations.

8. Further Comments
The time taken by the routine is approximately proportional to the number of evaluation points, m. The evaluation will be most efficient if the elements of PX are in non-decreasing order (or, more generally, if they are grouped in increasing order of the intervals [X(r-1),X(r)]). A single call of E01BGF with m>1 is more efficient than several calls with m=1.

9. Example
This example program reads in values of N, X, F and D, and calls E01BGF to compute the values of the interpolant and its derivative at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E01 -- Interpolation
E01BHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01BHF evaluates the definite integral of a piecewise cubic Hermite interpolant over the interval [a,b].

2. Specification

```fortran
SUBROUTINE E01BHF (N, X, F, D, A, B, PINT, IFAIL)
    INTEGER N, IFAIL
    DOUBLE PRECISION X(N), F(N), D(N), A, B, PINT
```

3. Description

This routine evaluates the definite integral of a piecewise cubic Hermite interpolant, as computed by E01BEF, over the interval [a,b].

If either a or b lies outside the interval from X(1) to X(N) computation of the integral involves extrapolation and a warning is returned.

The routine is derived from routine PCHIA in Fritsch [1].

4. References


5. Parameters

1:  N -- INTEGER  Input
2: X(N) -- DOUBLE PRECISION array Input
3: F(N) -- DOUBLE PRECISION array Input
4: D(N) -- DOUBLE PRECISION array Input
   On entry: N, X, F and D must be unchanged from the previous call of E01BEF.
5: A -- DOUBLE PRECISION Input
6: B -- DOUBLE PRECISION Input
   On entry: the interval [a,b] over which integration is to be performed.
7: PINT -- DOUBLE PRECISION Output
   On exit: the value of the definite integral of the interpolant over the interval [a,b].
8: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry N < 2.

IFAIL= 2
   The values of X(r), for r = 1,2,...,N, are not in strictly increasing order.

IFAIL= 3
   On entry at least one of A or B lies outside the interval [X(1),X(N)], and extrapolation was performed to compute the integral. The value returned is therefore unreliable.

7. Accuracy

The computational error in the value returned for PINT should be negligible in most practical situations.
8. Further Comments

The time taken by the routine is approximately proportional to
the number of data points included within the interval [a,b].

9. Example

This example program reads in values of N, X, F and D. It then
reads in pairs of values for A and B, and evaluates the definite
integral of the interpolant over the interval [A,B] until end-of-
file is reached.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E01 -- Interpolation  E01DAF
E01DAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01DAF computes a bicubic spline interpolating surface through a
set of data values, given on a rectangular grid in the x-y plane.

2. Specification

SUBROUTINE E01DAF (MX, MY, X, Y, F, PX, PY, LAMDA, MU, C,
1 WRK, IFAIL)
INTEGER MX, MY, PX, PY, IFAIL
DOUBLE PRECISION X(MX), Y(MY), F(MX*MY), LAMDA(MX+4), MU(MX
1 +4), C(MX*MY), WRK((MX+6)*(MY+6))

3. Description

This routine determines a bicubic spline interpolant to the set
of data points (x ,y ,f ), for q=1,2,...,m ; r=1,2,...,m . The
q r q,r x y
spline is given in the B-spline representation

\[ s(x,y) = \sum_{M} \sum_{N} c_{M}(x)N(y) \]
such that

\[
s(x, y) = f, \quad q_r q_r
\]

where \( M(x) \) and \( N(y) \) denote normalised cubic B-splines, the \( i \) \( j \) former defined on the knots \( (\lambda) \) to \( (\lambda) \) and the \( i \) \( i+4 \) latter on the knots \( (\mu) \) to \( (\mu) \), and the \( c_{ij} \) are the spline \( j \) \( j+4 \) coefficients. These knots, as well as the coefficients, are determined by the routine, which is derived from the routine B21RE in Anthony et al[1]. The method used is described in Section 8.2.


Values of the computed spline can subsequently be obtained by calling E02DEF or E02DFF as described in Section 8.3.

4. References


5. Parameters

1: \( MX \) -- INTEGER Input

2: \( MY \) -- INTEGER Input

On entry: \( MX \) and \( MY \) must specify \( m \) and \( m \) respectively, the number of points along the \( x \) and \( y \) axis that define the rectangular grid. Constraint: \( MX \geq 4 \) and \( MY \geq 4 \).
3: X(MX) -- DOUBLE PRECISION array  
   Input

4: Y(MY) -- DOUBLE PRECISION array  
   Input
   On entry: X(q) and Y(r) must contain x , for q=1,2,...,m ,  
   \( x \)
   and y , for r=1,2,...,m , respectively. Constraints:  
   \( y \)
   X(q) < X(q+1), for q=1,2,...,m -1,  
   \( x \)
   Y(r) < Y(r+1), for r=1,2,...,m -1.  
   \( y \)

5: F(MX*MY) -- DOUBLE PRECISION array  
   Input
   On entry: F(m *(q-1)+r) must contain f , for q=1,2,...,m ;  
   \( y \)
   \( q,r \)
   \( x \)
   r=1,2,...,m .  
   \( y \)

6: PX -- INTEGER  
   Output

7: PY -- INTEGER  
   Output
   On exit: PX and PY contain m +4 and m +4, the total number  
   \( x \)
   \( y \)
   of knots of the computed spline with respect to the x and y  
   variables, respectively.

8: LAMDA(MX+4) -- DOUBLE PRECISION array  
   Output
   On exit: LAMDA contains the complete set of knots (lambda)  
   \( i \)
   associated with the x variable, i.e., the interior knots  
   LAMDA(5), LAMDA(6), ..., LAMDA(PX-4), as well as the  
   additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) =  
   X(1) and LAMDA(PX-3) = LAMDA(PX-2) = LAMDA(PX-1) = LAMDA(PX)  
   = X(MX) needed for the B-spline representation. MU contains  
   the corresponding complete set of knots (mu) associated  
   \( i \)
   with the y variable.

9: MU(MY+4) -- DOUBLE PRECISION array  
   Output
   On exit: LAMDA contains the complete set of knots (lambda)  
   \( i \)
   associated with the x variable, i.e., the interior knots  
   LAMDA(5), LAMDA(6), ..., LAMDA(PX-4), as well as the  
   additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) =  
   X(1) and LAMDA(PX-3) = LAMDA(PX-2) = LAMDA(PX-1) = LAMDA(PX)  
   = X(MX) needed for the B-spline representation. MU contains  
   the corresponding complete set of knots (mu) associated  
   \( i \)
   with the y variable.

10: C(MX*MY) -- DOUBLE PRECISION array  
    Output
    On exit: the coefficients of the spline interpolant. C(  
    \( y \)
    m *(i-1)+j) contains the coefficient c described in  
    \( ij \)
    Section 3.

11: WRK((MX+6)*(MY+6)) -- DOUBLE PRECISION array  
    Workspace
12: IFAIL -- INTEGER
           Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
             familiar with this parameter (described in the Essential
             Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
             Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry MX < 4,
   or         MY < 4.

IFAIL= 2
   On entry either the values in the X array or the values in
   the Y array are not in increasing order.

IFAIL= 3
   A system of linear equations defining the B-spline
   coefficients was singular; the problem is too ill-
   conditioned to permit solution.

7. Accuracy

The main sources of rounding errors are in steps (2), (3), (6)
and (7) of the algorithm described in Section 8.2. It can be
shown (Cox [2]) that the matrix A formed in step (2) has
x elements differing relatively from their true values by at most a
small multiple of 3(epsilon), where (epsilon) is the machine
precision. A is 'totally positive', and a linear system with
x such a coefficient matrix can be solved quite safely by
elimination without pivoting. Similar comments apply to steps (6)
and (7). Thus the complete process is numerically stable.

8. Further Comments

8.1. Timing

The time taken by this routine is approximately proportional to
m m .
   x y
8.2. Outline of method used

The process of computing the spline consists of the following steps:

1. choice of the interior $x$-knots $(\lambda_i)_{5}^{m}$ as $(\lambda_i)_{i}^{=x_{i-2}}$, for $i=5,6,...,m$,

$$\lambda_{m-i} = x_{i-2},$$

2. formation of the system

$$A_x E = F,$$

where $A_x$ is a band matrix of order $m$ and bandwidth 4, containing in its $q$th row the values at $x$ of the $B$-splines $q$ in $x$, $F$ is the $m$ by $m$ rectangular matrix of values $f_{x,y}$, and $E$ denotes an $m$ by $m$ rectangular matrix of intermediate coefficients,

3. use of Gaussian elimination to reduce this system to band triangular form,

4. solution of this triangular system for $E$,

5. choice of the interior $y$ knots $(\mu_i)_{5}^{m}$ as $(\mu_i)_{i}^{=y_{i-2}}$, for $i=5,6,...,m$,

$$\mu_{m-i} = y_{i-2},$$

6. formation of the system

$$A_y C = E,$$

where $A_y$ is the counterpart of $A_x$ for the $y$ variable, and $C_y$ denotes the $m$ by $m$ rectangular matrix of values $c_{x,y}$,

7. use of Gaussian elimination to reduce this system to band triangular form,

8. solution of this triangular system for $C$ and hence $C$. 
For computational convenience, steps (2) and (3), and likewise steps (6) and (7), are combined so that the formation of $A$ and $x$ are carried out one row at a time.

8.3. Evaluation of Computed Spline

The values of the computed spline at the points $(TX(r),TY(r))$, for $r = 1,2,...,N$, may be obtained in the double precision array $FF$, of length at least $N$, by the following call:

```fortran
IFAIL = 0
CALL E02DEF(N,PX,PY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where $PX$, $PY$, $LAMDA$, $MU$ and $C$ are the output parameters of E01DAF, $WRK$ is a double precision workspace array of length at least $PY-4$, and $IWRK$ is an integer workspace array of length at least $PY-4$.

To evaluate the computed spline on an $NX$ by $NY$ rectangular grid of points in the $x$-$y$ plane, which is defined by the $x$ co-ordinates stored in $TX(q)$, for $q = 1,2,...,NX$, and the $y$ co-ordinates stored in $TY(r)$, for $r = 1,2,...,NY$, returning the results in the double precision array $FG$ which is of length at least $NX*NY$, the following call may be used:

```fortran
IFAIL = 0
CALL E02DFF(NX,NY,PX,PY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
             * IWRK,LIWRK,IFAIL)
```

where $PX$, $PY$, $LAMDA$, $MU$ and $C$ are the output parameters of E01DAF, $WRK$ is a double precision workspace array of length at least $LWRK = \min(NWRK1,NWRK2)$, $NWRK1 = NX*4+PX$, $NWRK2 = NY*4+PY$, and $IWRK$ is an integer workspace array of length at least $LIWRK = NY + PY - 4$ if $NWRK1 > NWRK2$, or $NX + PX - 4$ otherwise. The result of the spline evaluated at grid point $(q,r)$ is returned in element $(NY*(q-1)+r)$ of the array $FG$.

9. Example

This program reads in values of $m$, $x$ for $q=1,2,...,m$, $m$ and $y$ for $r=1,2,...,m$, followed by values of the ordinates $f_{q,r}$, defined at the grid points $(x,y)$. It then calls E01DAF to
compute a bicubic spline interpolant of the data values, and
prints the values of the knots and B-spline coefficients. Finally
it evaluates the spline at a small sample of points on a
rectangular grid.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E01 -- Interpolation
E01SAF -- NAG Foundation Library Routine Document
Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01SAF generates a two-dimensional surface interpolating a set of
scattered data points, using the method of Renka and Cline.

2. Specification

SUBROUTINE E01SAF (M, X, Y, F, TRIANG, GRADS, IFAIL)
INTEGER M, TRIANG(7*M), IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), GRADS(2,M)

3. Description

This routine constructs an interpolating surface \( F(x,y) \) through a
set of \( m \) scattered data points \((x_r, y_r, f_r)\), for \( r=1,2,\ldots,m \), using
\( r \)
a method due to Renka and Cline. In the \((x,y)\) plane, the data
points must be distinct. The constructed surface is continuous and has continuous first derivatives.

The method involves firstly creating a triangulation with all the
\((x,y)\) data points as nodes, the triangulation being as nearly
equiangular as possible (see Cline and Renka [1]). Then gradients
in the \( x \)- and \( y \)-directions are estimated at node \( r \), for
\( r=1,2,\ldots,m \), as the partial derivatives of a quadratic function
of \( x \) and \( y \) which interpolates the data value \( f_r \), and which fits
the data values at nearby nodes (those within a certain distance
chosen by the algorithm) in a weighted least-squares sense. The
weights are chosen such that closer nodes have more influence.
than more distant nodes on derivative estimates at node $r$. The computed partial derivatives, with the $f$ values, at the three $r$ nodes of each triangle define a piecewise polynomial surface of a certain form which is the interpolant on that triangle. See Renka and Cline [4] for more detailed information on the algorithm, a development of that by Lawson [2]. The code is derived from Renka [3].

The interpolant $F(x,y)$ can subsequently be evaluated at any point $(x,y)$ inside or outside the domain of the data by a call to E01SBF. Points outside the domain are evaluated by extrapolation.

4. References


5. Parameters

1:  M -- INTEGER Input  
    On entry: $m$, the number of data points. Constraint: $M \geq 3$.

2:  X(M) -- DOUBLE PRECISION array  
    Input

3:  Y(M) -- DOUBLE PRECISION array  
    Input

4:  F(M) -- DOUBLE PRECISION array  
    Input  
    On entry: the co-ordinates of the $r$th data point, for $r=1,2,\ldots,m$. The data points are accepted in any order, but see Section 8. Constraint: The $(x,y)$ nodes must not all be collinear, and each node must be unique.

5:  TRIANG(7*M) -- INTEGER array  
    Output  
    On exit: a data structure defining the computed triangulation, in a form suitable for passing to E01SBF.
6:  GRADS(2,M) -- DOUBLE PRECISION array
    On exit: the estimated partial derivatives at the nodes, in
    a form suitable for passing to E01SBF. The derivatives at
    node r with respect to x and y are contained in GRADS(1,r)
    and GRADS(2,r) respectively, for r=1,2,...,m.

7:  IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

   Errors detected by the routine:

   If on entry IFAIL = 0 or -1, explanatory error messages are
   output on the current error message unit (as defined by X04AAF).

   IFAIL= 1
       On entry M < 3.

   IFAIL= 2
       On entry all the (X,Y) pairs are collinear.

   IFAIL= 3
       On entry (X(i),Y(i)) = (X(j),Y(j)) for some i/=j.

7. Accuracy

   On successful exit, the computational errors should be negligible
   in most situations but the user should always check the computed
   surface for acceptability, by drawing contours for instance. The
   surface always interpolates the input data exactly.

8. Further Comments

   The time taken for a call of E01SAF is approximately proportional
   to the number of data points, m. The routine is more efficient
   if, before entry, the values in X, Y, F are arranged so that the
   X array is in ascending order.

9. Example

   This program reads in a set of 30 data points and calls E01SAF to
   construct an interpolating surface. It then calls E01SBF to
   evaluate the interpolant at a sample of points on a rectangular
Note that this example is not typical of a realistic problem: the number of data points would normally be larger, and the interpolant would need to be evaluated on a finer grid to obtain an accurate plot, say.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E01SBF evaluates at a given point the two-dimensional interpolant function computed by E01SAF.

This routine takes as input the parameters defining the interpolant \( F(x,y) \) of a set of scattered data points \( (x_r,y_r,f_r) \), for \( r=1,2,\ldots,m \), as computed by E01SAF, and evaluates the interpolant at the point \( (px,py) \).

If \( (px,py) \) is equal to \( (x_r,y_r) \) for some value of \( r \), the returned value will be equal to \( f_r \).

If \( (px,py) \) is not equal to \( (x_r,y_r) \) for any \( r \), the derivatives in \( (px,py) \) GRADS will be used to compute the interpolant. A triangle is sought which contains the point \( (px,py) \), and the vertices of the
triangle along with the partial derivatives and \( f \) values at the
vertices are used to compute the value \( F(px,py) \). If the point
\((px,py)\) lies outside the triangulation defined by the input
parameters, the returned value is obtained by extrapolation. In
this case, the interpolating function \( F \) is extended linearly
beyond the triangulation boundary. The method is described in
more detail in Renka and Cline [2] and the code is derived from
Renka [1].

E01SBF must only be called after a call to E01SAF.

4. References

Interpolation of Arbitrarily Distributed Points in the


5. Parameters

1: M -- INTEGER Input

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input

5: TRIANG(7*M) -- INTEGER array Input

6: GRADS(2,M) -- DOUBLE PRECISION array Input

On entry: M, X, Y, F, TRIANG and GRADS must be unchanged
from the previous call of E01SAF.

7: PX -- DOUBLE PRECISION Input

8: PY -- DOUBLE PRECISION Input

On entry: the point \((px,py)\) at which the interpolant is to
be evaluated.

9: PF -- DOUBLE PRECISION Output

On exit: the value of the interpolant evaluated at the
point \((px,py)\).

10: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 3.

IFAIL= 2
   On entry the triangulation information held in the array TRIANG does not specify a valid triangulation of the data points. TRIANG may have been corrupted since the call to E01SAF.

IFAIL= 3
   The evaluation point (PX,PY) lies outside the nodal triangulation, and the value returned in PF is computed by extrapolation.

7. Accuracy

Computational errors should be negligible in most practical situations.

8. Further Comments

The time taken for a call of E01SBF is approximately proportional to the number of data points, m.

The results returned by this routine are particularly suitable for applications such as graph plotting, producing a smooth surface from a number of scattered points.

9. Example

See the example for E01SAF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

E01SEF generates a two-dimensional surface interpolating a set of scattered data points, using a modified Shepard method.

2. Specification

```
SUBROUTINE E01SEF (M, X, Y, F, RNW, RNQ, NW, NQ, FNODES,
                   1 MINNQ, WRK, IFAIL)
   INTEGER M, NW, NQ, MINNQ, IFAIL
   DOUBLE PRECISION X(M), Y(M), F(M), RNW, RNQ, FNODES(5*M),
                   1 WRK(6*M)
```

3. Description

This routine constructs an interpolating surface $F(x, y)$ through a set of $m$ scattered data points $(x_r, y_r, f_r)$, for $r=1, 2, \ldots, m$, using a modification of Shepard's method. The surface is continuous and has continuous first derivatives.

The basic Shepard method, described in [2], interpolates the input data with the weighted mean

$$
F(x,y) = \frac{\sum_{r=1}^{m} \left( \frac{1}{d^2} \right) w(x,y)f_r}{\sum_{r=1}^{m} \left( \frac{1}{d^2} \right) w(x,y)}
$$

where $w(x,y) = \frac{1}{d^2}$ and $d = (x-x_r)^2 + (y-y_r)^2$. 

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
The basic method is global in that the interpolated value at any point depends on all the data, but this routine uses a modification due to Franke and Nielson described in [1], whereby the method becomes local by adjusting each \( w(x,y) \) to be zero outside a circle with centre \( (x,y) \) and some radius \( R \). Also, to improve the performance of the basic method, each \( f \) above is replaced by a function \( f(x,y) \), which is a quadratic fitted by weighted least-squares to data local to \( (x,y) \) and forced to interpolate \( (x,y,f) \). In this context, a point \( (x,y) \) is defined to be local to another point if it lies within some distance \( R \) of it. Computation of these quadratics constitutes the main work done by this routine. If there are less than 5 other points within distance \( R \) from \( (x,y) \), the quadratic is replaced by a linear function. In cases of rank-deficiency, the minimum norm solution is computed.

The user may specify values for \( R \) and \( R \), but it is usually easier to choose instead two integers \( N \) and \( N \), from which the routine will compute \( R \) and \( R \). These integers can be thought of as the average numbers of data points lying within distances \( R \) and \( R \) respectively from each node. Default values are provided, and advice on alternatives is given in Section 8.2.

The interpolant \( F(x,y) \) generated by this routine can subsequently be evaluated for any point \( (x,y) \) in the domain of the data by a call to E01SFF.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of data points. Constraint: M >= 3.

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input
   On entry: the co-ordinates of the rth data point, for
   r=1,2,...,m. The order of the data points is immaterial.
   Constraint: each of the (X(r),Y(r)) pairs must be unique.

5: RNW -- DOUBLE PRECISION Input/Output

6: RNQ -- DOUBLE PRECISION Input/Output
   On entry: suitable values for the radii R and R ,
   w q
   described in Section 3. Constraint: RNQ <= 0 or 0 < RNW <=
   RNQ. On exit: if RNQ is set less than or equal to zero on
   entry, then default values for both of them will be computed
   from the parameters NW and NQ, and RNW and RNQ will contain
   these values on exit.

7: NW -- INTEGER Input

8: NQ -- INTEGER Input
   On entry: if RNQ > 0.0 and RNW > 0.0 then NW and NQ are not
   referenced by the routine. Otherwise, NW and NQ must specify
   suitable values for the integers N and N described in
   Section 3.

   If NQ is less than or equal to zero on entry, then default
   values for both of them, namely NW = 9 and NQ = 18, will be
   used. Constraint: NQ <= 0 or 0 < NW <= NQ.

9: FNODES(5*M) -- DOUBLE PRECISION array Output
   On exit: the coefficients of the constructed quadratic
   nodal functions. These are in a form suitable for passing to
   E01SFF.

10: MINNQ -- INTEGER Output
    On exit: the minimum number of data points that lie within
    radius RNQ of any node, and thus define a nodal function. If
    MINNQ is very small (say, less than 5), then the interpolant
    may be unsatisfactory in regions where the data points are
    sparse.
11: WRK(6*M) -- DOUBLE PRECISION array
Workspace

12: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators andWarnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry M < 3.

IFAIL= 2
On entry RNQ > 0 and either RNW > RNQ or RNW <= 0.

IFAIL= 3
On entry NQ > 0 and either NW > NQ or NW <= 0.

IFAIL= 4
On entry (X(i),Y(i)) is equal to (X(j),Y(j)) for some i/=j.

7. Accuracy

On successful exit, the computational errors should be negligible
in most situations but the user should always check the computed
surface for acceptability, by drawing contours for instance. The
surface always interpolates the input data exactly.

8. Further Comments

8.1. Timing

The time taken for a call of E01SEF is approximately proportional
to the number of data points, m, provided that q
order as its default value (18). However if N is increased so
that the method becomes more global, the time taken becomes
2
approximately proportional to m .

m N} ${\rm w}}$ and ${\rm N} {\rm q}}$
8.2. Choice of $\{$

Note first that the radii $R_w$ and $R_q$, described in Section 3, are computed as $D_N$ and $D_quad$, respectively, where $D$ is the maximum distance between any pair of data points. Default values $N_w=9$ and $N_q=18$ work quite well when the data points are fairly uniformly distributed. However, for data having some regions with relatively few points or for small data sets ($m<25$), a larger value of $N$ may be needed. This is to ensure a reasonable number of data points within a distance $R$ of each node, and to avoid some regions in the data area being left outside all the discs of radius $R$ on which the weights $w(x,y)$ are non-zero. Maintaining $N$ approximately equal to $2N_q$ is usually an advantage.

Note however that increasing $N_w$ and $N_q$ does not improve the quality of the interpolant in all cases. It does increase the computational cost and makes the method less local.

9. Example

This program reads in a set of 30 data points and calls E01SEF to construct an interpolating surface. It then calls E01SFF to evaluate the interpolant at a sample of points on a rectangular grid.

Note that this example is not typical of a realistic problem: the number of data points would normally be larger, and the interpolant would need to be evaluated on a finer grid to obtain an accurate plot, say.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01SFF evaluates at a given point the two-dimensional interpolating function computed by E01SEF.

2. Specification

```fortran
SUBROUTINE E01SFF (M, X, Y, F, RNW, FNODES, PX, PY, PF,
                    IFAIL)
 INTEGER M, IFAIL
 DOUBLE PRECISION X(M), Y(M), F(M), RNW, FNODES(5*M), PX,
                    PY, PF
```

3. Description

This routine takes as input the interpolant $F(x,y)$ of a set of scattered data points $(x_r, y_r, f_r)$, for $r = 1, 2, \ldots, m$, as computed by E01SEF, and evaluates the interpolant at the point $(px, py)$.

If $(px, py)$ is equal to $(x_r, y_r)$ for some value of $r$, the returned value will be equal to $f_r$.

If $(px, py)$ is not equal to $(x_r, y_r)$ for any $r$, all points that are within distance RNW of $(px, py)$, along with the corresponding nodal functions given by FNODES, will be used to compute a value of the interpolant.

E01SFF must only be called after a call to E01SEF.

4. References


5. Parameters
1: M -- INTEGER Input
2: X(M) -- DOUBLE PRECISION array Input
3: Y(M) -- DOUBLE PRECISION array Input
4: F(M) -- DOUBLE PRECISION array Input
5: RNW -- DOUBLE PRECISION Input
6: FNODES(5*M) -- DOUBLE PRECISION array Input
   On entry: M, X, Y, F, RNW and FNODES must be unchanged from
   the previous call of E01SEF.
7: PX -- DOUBLE PRECISION Input
8: PY -- DOUBLE PRECISION Input
   On entry: the point (px,py) at which the interpolant is to
   be evaluated.
9: PF -- DOUBLE PRECISION Output
   On exit: the value of the interpolant evaluated at the
   point (px,py).
10: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 3.

IFAIL= 2
   The interpolant cannot be evaluated because the evaluation
   point (PX,PY) lies outside the support region of the data
   supplied in X, Y and F. This error exit will occur if
   (PX,PY) lies at a distance greater than or equal to RNW from
   every point given by arrays X and Y.
The value 0.0 is returned in PF. This value will not provide continuity with values obtained at other points (PX, PY), i.e., values obtained when IFAIL = 0 on exit.

7. Accuracy

Computational errors should be negligible in most practical situations.

8. Further Comments

The time taken for a call of E01SFF is approximately proportional to the number of data points, m.

The results returned by this routine are particularly suitable for applications such as graph plotting, producing a smooth surface from a number of scattered points.

9. Example

See the example for E01SEF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

NagInterpolationPackage (NAGE01)

Exports:
e01baf e01bef e01bff e01bgf e01bhf
e01daf e01saf e01sbf e01sef e01sff
package NAGE01 NagInterpolationPackage

)abbrev package NAGE01 NagInterpolationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:53 1994
++ This package uses the NAG Library to calculate the interpolation of a
++ function of one or two variables. When provided with the value of the
++ function (and possibly one or more of its lowest-order
++ derivatives) at each of a number of values of the variable(s),
++ the routines provide either an interpolating function or an
++ interpolated value. For some of the interpolating functions,
++ there are supporting routines to evaluate, differentiate or
++ integrate them.

NagInterpolationPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage
Exports == with
e01baf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_,
  Integer,Integer) -> Result
++ e01baf(m,x,y,lck,lwrk,ifail)
++ determines a cubic spline to a given set of
++ data.
++ See \downlink{Manual Page}{manpageXXe01baf}.
e01bef : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ e01bef(n,x,f,ifail)
++ computes a monotonicity-preserving piecewise cubic Hermite
++ interpolant to a set of data points.
++ See \downlink{Manual Page}{manpageXXe01bef}.
e01bff: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_,
  Integer,Matrix DoubleFloat,Integer) -> Result
++ e01bff(n,x,f,d,m,px,ifail)
++ evaluates a piecewise cubic Hermite interpolant at a set
++ of points.
++ See \downlink{Manual Page}{manpageXXe01bff}.
e01bgf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_,
  Integer,Matrix DoubleFloat,Integer) -> Result
++ e01bgf(n,x,f,d,m,px,ifail)
++ evaluates a piecewise cubic Hermite interpolant and its
++ first derivative at a set of points.
++ See \downlink{Manual Page}{manpageXXe01bgf}.
e01bhf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_,
  DoubleFloat,DoubleFloat,Integer) -> Result
++ e01bhf(n,x,f,d,a,b,ifail)
++ evaluates the definite integral of a piecewise cubic
++ Hermite interpolant over the interval [a,b].
++ See \downlink{Manual Page}{manpageXXe01bhf}.

e01daf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
++ e01daf(mx,my,x,y,f,ifail)
++ computes a bicubic spline interpolating surface through a
++ set of data values, given on a rectangular grid in the x-y plane.
++ See \downlink{Manual Page}{manpageXXe01daf}.

++ See \downlink{Manual Page}{manpageXXe01saf}.

e01saf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer) -> Result
++ e01saf(m,x,y,f,ifail)
++ generates a two-dimensional surface interpolating a set of
++ scattered data points, using the method of Renka and Cline.
++ See \downlink{Manual Page}{manpageXXe01saf}.

++ See \downlink{Manual Page}{manpageXXe01sbf}.

e01sbf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_,
        Matrix Integer,Matrix DoubleFloat,DoubleFloat,DoubleFloat,_,
        Integer) -> Result
++ e01sbf(m,x,y,f,triang,grads,px,py,ifail)
++ evaluates at a given point the two-dimensional interpolant
++ function computed by E01SAF.
++ See \downlink{Manual Page}{manpageXXe01sbf}.

++ See \downlink{Manual Page}{manpageXXe01sef}.

e01sef: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer,Integer,DoubleFloat,DoubleFloat,Integer) -> Result
++ e01sef(m,x,y,f,nw,nq,rnw,rnq,ifail)
++ generates a two-dimensional surface interpolating a set of
++ scattered data points, using a modified Shepard method.
++ See \downlink{Manual Page}{manpageXXe01sef}.

++ See \downlink{Manual Page}{manpageXXe01sff}.

e01sff: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        DoubleFloat,Matrix DoubleFloat,DoubleFloat,DoubleFloat,_,
        Integer) -> Result
++ e01sff(m,x,y,f,nw,fnodes,px,py,ifail)
++ evaluates at a given point the two-dimensional
++ interpolating function computed by E01SEF.
++ See \downlink{Manual Page}{manpageXXe01sff}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)
import AnyFunctions1(DoubleFloat)

e01baf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
[(invokeNagman(NIL$Lisp, _
"e01baf", _
"lamda"::S,"c"::S,"wrk"::S]_ _
$Lisp, _
["lamda"::S,"c"::S,"wrk"::S]$Lisp, _
["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp , _
["wrk"::S,"lwrk"::S]$Lisp]$Lisp _ _
$Lisp, _
]$List, _
["lamda"::S,"c"::S,"ifail"::S]$Lisp, _
[(mArg::Any,lckArg::Any,lwrkArg::Any,ifailArg::Any, _
xArg::Any,yArg::Any ] _ _
@List Any]$Lisp)$Lisp) _ _
pretend List (Record(key:Symbol,entry:Any))$Result

e01bef(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"e01bef", _
["d"::S]$Lisp, _
["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp , _
["d"::S,"n"::S]$Lisp]$Lisp]$Lisp _ _
["integer"::S,"n"::S,"ifail"::S]$Lisp _ _
]$List, _
["d"::S,"ifail"::S]$Lisp, _
[(nArg::Any,ifailArg::Any,xArg::Any,fArg::Any ] _ _
@List Any]$Lisp)$Lisp) _ _
pretend List (Record(key:Symbol,entry:Any))$Result

e01bff(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
dArg:Matrix DoubleFloat,mArg:Integer,pxArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"e01bff", _
]_ _
$Lisp, _
["pf"::S]$Lisp, _
["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp , _
["d"::S,"n"::S]$Lisp,["px"::S,"m"::S]$Lisp, _
["pf"::S,"m"::S]$Lisp]$Lisp]$Lisp _ _
["integer"::S,"n"::S,"m"::S,"ifail"::S]$Lisp _ _
]$List, _
["pf"::S,"ifail"::S]$Lisp, _
[(nArg::Any,mArg::Any,ifailArg::Any,xArg::Any,fArg::Any, _
dArg::Any,pxArg::Any ] _ _
)
e01bhf(nArg: Integer, xArg: Matrix DoubleFloat, fArg: Matrix DoubleFloat, 
 dArg: Matrix DoubleFloat, aArg: DoubleFloat, bArg: DoubleFloat, 
 ifailArg: Integer): Result ==
 [(invokeNagman(NIL$Lisp, 
  "e01bhf", 
  ["x":S,"n":S,"f":S,"d":S,"pf":S, 
  "pd":S]$Lisp, 
  ["double":S,"x":S,"n":S]$Lisp, ["f":S,"n":S]$Lisp, 
  , ["double":S,"n":S]$Lisp] $Lisp, 
  , ["double":S,"m":S]$Lisp, ["pf":S,"pd":S]$Lisp, 
  ]$Lisp_ 
  ["pf":S,"pd":S,"ifail":S]$Lisp_ 
  )
)]@List Any$Lisp)$Lisp) 

pretend List (Record(key:Symbol,entry:Any))$Result

\[\text{e01saf}(mArg:\text{Integer}, xArg: \text{Matrix DoubleFloat}, yArg: \text{Matrix DoubleFloat}, fArg: \text{Matrix DoubleFloat}, ifailArg: \text{Integer}): \text{Result} =\]
\[
\text{pretend List (Record(key:Symbol,entry:Any))} \text{Result}
\]

\[\text{e01sbf}(mArg:\text{Integer}, xArg: \text{Matrix DoubleFloat}, yArg: \text{Matrix DoubleFloat},
\quad fArg: \text{Matrix DoubleFloat}, triangArg: \text{Matrix Integer},
\quad gradsArg: \text{Matrix DoubleFloat}, pxArg: \text{DoubleFloat}, pyArg: \text{DoubleFloat},
\quad ifailArg: \text{Integer}): \text{Result} =\]
\[
\text{pretend List (Record(key:Symbol,entry:Any))} \text{Result}
\]
\[
[(\text{list} \text{any}, \text{pxArg::Any, pyArg::Any}, \text{ifailArg::Any}, x\text{Arg::Any,}_
\text{yArg::Any, fArg::Any, triangArg::Any, gradsArg::Any})]
\]

\[
\text{pretend list (record (key: Symbol, entry: Any))}]\text{Result}
\]

\[
e01sef(\text{mArg::Integer, xArg::Matrix DoubleFloat, yArg::Matrix DoubleFloat,}_
\text{fArg::Matrix DoubleFloat, nwArg::Integer, nqArg::Integer,}_
\text{rnwArg::DoubleFloat, rqnArg::DoubleFloat, ifailArg::Integer}): \text{Result} ==
\]

\[
[(\text{invokeNagman(NIL$Lisp,}
\text{"e01sef",}
["m":S, "nw":S, "nq":S, "minnq":S, "rnw":S,}
]\text{Lisp}_
\]

\[
[(\text{list any}, \text{nwArg::Any, nqArg::Any, rnwArg::Any,}
\text{rnqArg::Any, ifailArg::Any, xArg::Any, yArg::Any, fArg::Any})]
\]

\[
\text{pretend list (record (key: Symbol, entry: Any))}]\text{Result}
\]

\[
e01sff(\text{mArg::Integer, xArg::Matrix DoubleFloat, yArg::Matrix DoubleFloat,}_
\text{fArg::Matrix DoubleFloat, nwArg::DoubleFloat, xArg::DoubleFloat,}_
\text{yArg::Matrix DoubleFloat, ifailArg::Integer}): \text{Result} ==
\]

\[
[(\text{invokeNagman(NIL$Lisp,}
\text{"e01sff",}
["m":S, "rnw":S, "px":S, "py":S, "pf":S,}
,"ifail":S, "x":S, "y":S, "f":S, "fnodes":S]
]\text{Lisp}_
\]

\[
[(\text{list any}, \text{rnwArg::Any, pxArg::Any, pyArg::Any,}
\text{ifailArg::Any, xArg::Any, yArg::Any, fArg::Any, fnodesArg::Any})]
\]

\[
\text{pretend list (record (key: Symbol, entry: Any))}]\text{Result}
\]
package NAGF07 NagLapack

--- NagLapack.input ---

)set break resume
)sys rm -f NagLapack.output
)spool NagLapack.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagLapack
--E 1

)spool
)lisp (bye)

--- NagLapack.help ---

This package uses the NAG Library to compute matrix factorizations, and to solve systems of linear equations following the matrix factorizations.

F07 -- Linear Equations (LAPACK)  Introduction -- F07
Chapter F07
Linear Equations (LAPACK)

1. Scope of the Chapter

This chapter provides four routines concerned with matrix factorization, and the solution of systems of linear equations following the matrix factorizations.
2. Background to the Problems

Background material, together with pointers to the routines in this chapter, are to be found in the F01 and F04 Chapter Introductions.

3. Recommendations on Choice and Use of Routines

The routines in this chapter are derived from the LAPACK project and may also be called using the LAPACK name, which is given in brackets following the F07 name in the following descriptions.

Routine F07ADF (DGETRF) performs an LU factorization of a real m by n matrix A. Following the use of this routine, F07AEF (DGETRS) may be used to solve a system of n non-singular linear equations, with one or more right-hand sides.

Routine F07FDF (DPOTRF) performs the Cholesky factorization of a real symmetric positive-definite matrix A. Following the use of this routine, F07FEF (DPOTRS) may be used to solve a system of symmetric positive-definite linear equations, with one or more right-hand sides.

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F07ADF (DGETRF) computes the LU factorization of a real m by n matrix.

2. Specification

```fortran
SUBROUTINE F07ADF (M, N, A, LDA, IPIV, INFO)
ENTRY M, N, A, LDA, IPIV, INFO
INTEGER M, N, LDA, IPIV(*), INFO
DOUBLE PRECISION A(LDA,*)
```

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

This routine forms the LU factorization of a real m by n matrix A as A=PLU, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m>n) and U is upper triangular (upper trapezoidal if m<n). Usually A is square (m=n), and both L and U are triangular. The routine uses partial pivoting, with row interchanges.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of rows of the matrix A.
   Constraint: M >= 0.

2: N -- INTEGER Input
   On entry: n, the number of columns of the matrix A.
   Constraint: N >= 0.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the m by n matrix A. On exit: A is overwritten by the factors L and U; the unit diagonal elements of L are not stored.
4: LDA -- INTEGER
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which F07ADF is called.
   Constraint: LDA >= max(1,M).

5: IPIV(*) -- INTEGER array
   Note: the dimension of the array IPIV must be at least
   max(1,min(M,N)).
   On exit: the pivot indices. Row i of the matrix A was
   interchanged with row IPIV(i) for i=1,2,...,min(m,n).

6: INFO -- INTEGER
   On exit: INFO = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

INFO < 0
   If INFO = -i, the ith parameter has an illegal value. An
   explanatory message is output, and execution of the program
   is terminated.

INFO > 0
   If INFO = i, u is exactly zero. The factorization has been
   completed but the factor U is exactly singular, and division
   by zero will occur if it is subsequently used to solve a
   system of linear equations or to compute A^{-1}.

7. Accuracy

The computed factors L and U are the exact factors of a perturbed
matrix A+E, where

|E| <= c(min(m,n))(epsilon)P|L||U|,

C(n) is a modest linear function of n, and (epsilon) is the
machine precision.

8. Further Comments

The total number of floating-point operations is approximately

\[ 2 \times 3^{1/3} 2 \times 12 \times 12 \]

-n if m=n (the usual case), -n (3m-n) if m>n and
-n (3n-m) if m<n.

A call to this routine with m=n may be followed by calls to the
routines:

\[
T
\]

F07AEF (DGETRS) to solve AX=B or A X=B;

F07AGF (DGECON)(*) to estimate the condition number of A;

F07AJF (DGETRI)(*) to compute the inverse of A.

The complex analogue of this routine is F07ARF (ZGETRF)(*).

9. Example

To compute the LU factorization of the matrix A, where

\[
\begin{pmatrix}
1.80 & 2.88 & 2.05 & -0.89 \\
5.25 & -2.95 & -0.95 & -3.80 \\
1.58 & -2.69 & -2.90 & -1.04 \\
-1.11 & -0.66 & -0.59 & 0.80
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

To solve a real system of linear equations $AX=B$ or $A^T X=B$, this routine must be preceded by a call to F07ADF (DGETRF) which computes the LU factorization of $A$ as $A=PLU$. The solution is computed by forward and backward substitution.

If $TRANS = 'N'$, the solution is computed by solving $PLY=B$ and then $UX=Y$.

If $TRANS = 'T'$ or $'C'$, the solution is computed by solving $U^TY=B$ and then $L^TPX=Y$.

4. References


5. Parameters

1: TRANS -- CHARACTER*1 Input
   On entry: indicates the form of the equations as follows:
   
   if $TRANS = 'N'$, then $AX=B$ is solved for $X$;

   if $TRANS = 'T'$ or $'C'$, then $A^T X=B$ is solved for $X$.

   Constraint: $TRANS = 'N'$, $'T'$ or $'C'$.

2: N -- INTEGER Input
   On entry: $n$, the order of the matrix $A$. Constraint: $N >= 0$.

3: NRHS -- INTEGER Input
   On entry: $r$, the number of right-hand sides. Constraint: $NRHS >= 0$.

4: A(LDA,*) -- DOUBLE PRECISION array Input
   Note: the second dimension of the array A must be at least
   \[ \max(1,N) \].
   On entry: the LU factorization of $A$, as returned by F07ADF (DGETRF).

5: LDA -- INTEGER Input
   On entry:
the first dimension of the array A as declared in the
(sub)program from which F07AEF is called.
Constraint: LDA >= max(1,N).

6: IPIV(*) -- INTEGER array
   Input
   Note: the dimension of the array IPIV must be at least
   max(1,N).
   On entry: the pivot indices, as returned by F07ADF (DGETRF).

7: B(LDB,*) -- DOUBLE PRECISION array
   Input/Output
   Note: the second dimension of the array B must be at least
   max(1,NRHS).
   On entry: the n by r right-hand side matrix B. On exit: the
   n by r solution matrix X.

8: LDB -- INTEGER
   Input
   On entry:
   the first dimension of the array B as declared in the
   (sub)program from which F07AEF is called.
   Constraint: LDB >= max(1,N).

9: INFO -- INTEGER
   Output
   On exit: INFO = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

INFO < 0
If INFO = -i, the ith parameter has an illegal value. An
explanatory message is output, and execution of the program
is terminated.

7. Accuracy

For each right-hand side vector b, the computed solution x is the
exact solution of a perturbed system of equations (A+E)x=b, where

\[ |E| \leq c(n)(\text{epsilon})P|L||U|, \]

\( c(n) \) is a modest linear function of \( n \), and \((\text{epsilon})\) is the
machine precision.

\[ \text{infy} \]

If \( x \) is the true solution, then the computed solution \( x \) satisfies
a forward error bound of the form

\[ \frac{||x-x||}{\text{infy}} \leq c(n)\text{cond}(A,x)(\text{epsilon}) \]
\[ \|\|x\|\|_{\infty}^{-1} \leq \text{cond}(A, x) = \|\|A\|\|_{\infty} \|\|x\|\|_{\infty} \text{ / } \|\|x\|\|_{\infty}^{-1} \leq \text{cond}(A) = \|\|A\|\|_{\infty} \text{ / } \|\|A\|\|_{\infty}^{-1} \leq \kappa(A). \]

Note that \( \text{cond}(A, x) \) can be much smaller than \( \text{cond}(A) \), and \( \text{cond}(A^T) \) can be much larger (or smaller) than \( \text{cond}(A) \).

Forward and backward error bounds can be computed by calling F07AHF (DGERFS)(*), and an estimate for \( \kappa(A) \) can be obtained by calling F07AGF (DGECON)(*) with NORM = 'I'.

8. Further Comments

The total number of floating-point operations is approximately \( 2 \cdot 2n^3 \).

This routine may be followed by a call to F07AHF (DGERFS)(*), and return the error estimate.

The complex analogue of this routine is F07ASF (ZGETRS)(*).

9. Example

To solve the system of equations \( AX = B \), where

\[
\begin{pmatrix} 1.80 & 2.88 & 2.05 & -0.89 \\ 5.25 & -2.95 & -0.95 & -3.80 \\ 1.58 & -2.69 & -2.90 & -1.04 \\ -1.11 & -0.66 & -0.59 & 0.80 \end{pmatrix} \times \begin{pmatrix} 9.52 & 18.47 \\ 24.35 & 2.25 \\ 0.77 & -13.28 \\ -6.22 & -6.21 \end{pmatrix} = \begin{pmatrix} 1.80 & 2.88 & 2.05 & -0.89 \\ 5.25 & -2.95 & -0.95 & -3.80 \\ 1.58 & -2.69 & -2.90 & -1.04 \\ -1.11 & -0.66 & -0.59 & 0.80 \end{pmatrix}
\]

and

\[
\begin{pmatrix} 9.52 & 18.47 \\ 24.35 & 2.25 \\ 0.77 & -13.28 \\ -6.22 & -6.21 \end{pmatrix}
\]

Here \( A \) is unsymmetric and must first be factorized by F07ADF (DGETRF)).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F07 -- Linear Equations (LAPACK)

F07FDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F07FDF (DPOTRF) computes the Cholesky factorization of a real
symmetric positive-definite matrix.

2. Specification

SUBROUTINE F07FDF (UPLO, N, A, LDA, INFO)
ENTRY UPLO, N, A, LDA, INFO
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
CHARACTER*1 UPLO

The ENTRY statement enables the routine to be called by its
LAPACK name.

3. Description

This routine forms the Cholesky factorization of a real symmetric
positive-definite matrix \( A \) either as \( A = U^T U \) if \( \text{UPLO} = 'U' \) or \( A = LL^T \)
if \( \text{UPLO} = 'L' \), where \( U \) is an upper triangular matrix and \( L \) is
lower triangular.

4. References

LAPACK Working Note No. 14. University of Tennessee,
Knoxville.

Edition). Johns Hopkins University Press, Baltimore,
Maryland.

5. Parameters

1: UPLO -- CHARACTER*1  Input
On entry: indicates whether the upper or lower triangular part of A is stored and how A is factorized, as follows:
   if UPLO = 'U', then the upper triangular part of A is stored and A is factorized as $U^T U$, where $U$ is upper triangular;
   if UPLO = 'L', then the lower triangular part of A is stored and A is factorized as $L^T L$, where $L$ is lower triangular.
Constraint: UPLO = 'U' or 'L'.

2: N -- INTEGER Input
On entry: n, the order of the matrix A. Constraint: N >= 0.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array A must be at least max(1,N).
On entry: the n by n symmetric positive-definite matrix A. If UPLO = 'U', the upper triangle of A must be stored and the elements of the array below the diagonal are not referenced; if UPLO = 'L', the lower triangle of A must be stored and the elements of the array above the diagonal are not referenced. On exit: the upper or lower triangle of A is overwritten by the Cholesky factor $U$ or $L$ as specified by UPLO.

4: LDA -- INTEGER Input
On entry:
   the first dimension of the array A as declared in the (sub)program from which F07FDF is called.
Constraint: LDA >= max(1,N).

5: INFO -- INTEGER Output
On exit: INFO = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

INFO < 0
   If INFO = -i, the ith parameter has an illegal value. An explanatory message is output, and execution of the program is terminated.

INFO > 0
   If INFO = i, the leading minor of order i is not positive-definite and the factorization could not be completed. Hence A itself is not positive-definite. This may indicate an error in forming the matrix A. To factorize a symmetric
7. Accuracy

If UPLO = 'U', the computed factor U is the exact factor of a perturbed matrix A+E, where

\[ |E| \leq c(n)(\text{epsilon})|U||U|, \]

\[ c(n) \text{ is a modest linear function of } n, \text{ and (epsilon) is the machine precision. If UPLO = 'L', a similar statement holds for the computed factor L. It follows that} \]

\[ |e| \leq c(n)(\text{epsilon}) /a_{ij}/a_{ii}, \]

\[ \forall \quad ii\quad jj \]

8. Further Comments

The total number of floating-point operations is approximately

\[ 13n^3. \]

A call to this routine may be followed by calls to the routines:

- F07FEF (DPOTRS) to solve AX=B;
- F07FGF (DPOCON)(*) to estimate the condition number of A;
- F07FJF (DPOTRI)(*) to compute the inverse of A.

The complex analogue of this routine is F07FRF (ZPOTRF)(*).

9. Example

To compute the Cholesky factorization of the matrix A, where

\[
\begin{pmatrix}
4.16 & -3.12 & 0.56 & -0.10 \\
-3.12 & 5.03 & -0.83 & 1.18 \\
0.56 & -0.83 & 0.76 & 0.34 \\
-0.10 & 1.18 & 0.34 & 1.18 \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F07 -- Linear Equations (LAPACK)

F07FEF

F07FEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F07FEF (DPOTRS) solves a real symmetric positive-definite system of linear equations with multiple right-hand sides, AX=B, where A has been factorized by F07FDF (DPOTRF).

2. Specification

SUBROUTINE F07FEF (UPLO, N, NRHS, A, LDA, B, LDB, INFO)
ENTRY UPLO, N, NRHS, A, LDA, B, LDB, INFO
INTEGER N, NRHS, LDA, B, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
CHARACTER*1 UPLO

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

To solve a real symmetric positive-definite system of linear equations AX=B, this routine must be preceded by a call to F07FDF (DPOTRF) which computes the Cholesky factorization of A. The solution X is computed by forward and backward substitution.

T
If UPLO = 'U', A=U U, where U is upper triangular; the solution X
T
is computed by solving U Y=B and then UX=Y.

T
If UPLO = 'L', A=LL, where L is lower triangular; the solution X
T
is computed by solving LY=B and then L X=Y.

4. References


5. Parameters
1: UPLO -- CHARACTER*1  
   On entry: indicates whether the upper or lower triangular  
   part of A is stored and how A is factorized, as follows:  
   \[
   \begin{align*}
   &T \\
   &\text{if UPLO = 'U', then } A = U^\top U \text{ where } U \text{ is upper triangular}; \\
   &T \\
   &\text{if UPLO = 'L', then } A = L L^\top \text{ where } L \text{ is lower triangular.} 
   \end{align*}
   \]
   Constraint: UPLO = 'U' or 'L'.

2: N -- INTEGER  
   On entry: n, the order of the matrix A. Constraint: N \geq 0.

3: NRHS -- INTEGER  
   On entry: r, the number of right-hand sides. Constraint: NRHS \geq 0.

4: A(LDA,*) -- DOUBLE PRECISION array  
   Note: the second dimension of the array A must be at least  
   \(\max(1,N)\).
   On entry: the Cholesky factor of A, as returned by F07FDF  
   (DPOTRF).

5: LDA -- INTEGER  
   On entry:  
   the first dimension of the array A as declared in the  
   (sub)program from which F07FEF is called. 
   Constraint: LDA \geq \max(1,N).

6: B(LDB,*) -- DOUBLE PRECISION array  
   Input/Output  
   Note: the second dimension of the array B must be at least  
   \(\max(1,NRHS)\).
   On entry: the n by r right-hand side matrix B.

7: LDB -- INTEGER  
   On entry:  
   the first dimension of the array B as declared in the  
   (sub)program from which F07FEF is called. 
   Constraint: LDB \geq \max(1,N).

8: INFO -- INTEGER  
   Output  
   On exit: INFO = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

INFO \leq 0  
If INFO = -i, the i\textsuperscript{th} parameter has an illegal value. An  
explanatory message is output, and execution of the program
7. Accuracy

For each right-hand side vector $b$, the computed solution $x$ is the exact solution of a perturbed system of equations $(A+E)x=b$, where

$$
|E| \leq c(n)(\epsilon)|U||U| \text{ if } \text{UPLO} = 'U',
$$

$$
|E| \leq c(n)(\epsilon)|L||L| \text{ if } \text{UPLO} = 'L',
$$

$c(n)$ is a modest linear function of $n$, and $(\epsilon)$ is the machine precision.

If $x$ is the true solution, then the computed solution $x$ satisfies a forward bound of the form

$$
\frac{||x-x||_{\infty}}{||x||_{\infty}} \leq c(n)\text{cond}(A,x)(\epsilon)
$$

where $\text{cond}(A,x) = \frac{|||A||A||x||}{||x||}$.

Further and backward error bounds can be computed by calling F07FHF (DPORFS)(*), and an estimate for $(\kappa)(A)$ can be obtained by calling F07FGF (DPOCON)(*).

8. Further Comments

The total number of floating-point operations is approximately $2n^2$.

This routine may be followed by a call to F07FHF (DPORFS)(*).
The complex analogue of this routine is F07FSF (ZPOTRS)(*).

9. Example

To compute the Cholesky factorization of the matrix \( A \), where

\[
A = \begin{pmatrix}
4.16 & -3.12 & 0.56 & -0.10 \\
-3.12 & 5.03 & -0.83 & 1.18 \\
0.56 & -0.83 & 0.76 & 0.34 \\
-0.10 & 1.18 & 0.34 & 1.18
\end{pmatrix}
\]

and

\[
B = \begin{pmatrix}
8.70 & 8.30 \\
-13.35 & 2.13 \\
1.89 & 1.61 \\
-4.14 & 5.00
\end{pmatrix}
\]

Here \( A \) is symmetric positive-definite and must first be
factorized by F07FDF (DPOTRF).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

NagLapack (NAGF07)

Exports:
f07adf  f07aef  f07fdf  f07fef

— package NAGF07 NagLapack —
)abbrev package NAGF07 NagLapack
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:42 1994
++ Description:
++ This package uses the NAG Library to compute matrix
++ factorizations, and to solve systems of linear equations
++ following the matrix factorizations.

NagLapack():Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports == with
  f07adf : (Integer,Integer,Integer,Matrix DoubleFloat) -> Result
  ++ f07adf(m,n,lda,a)
  ++ (DGETRF) computes the LU factorization of a real m by n
  ++ matrix.
  ++ See \downlink{Manual Page}{manpageXXf07adf}.
  f07aef : (String,Integer,Integer,Matrix DoubleFloat,_,
       Integer,Matrix Integer,Integer,Matrix DoubleFloat) -> Result
  ++ f07aef(trans,n,nrhs,a,lda,ipiv,ldb,b)
  ++ (DGETRS) solves a real system of linear equations with
  ++ multiple right-hand sides, AX=B or A X=B, where A has been
  ++ factorized by F07ADF (DGETRF).
  ++ See \downlink{Manual Page}{manpageXXf07aef}.
  f07fdf : (String,Integer,Integer,Matrix DoubleFloat) -> Result
  ++ f07fdf(uplo,n,lda,a)
  ++ (DPOTRF) computes the Cholesky factorization of a real
  ++ symmetric positive-definite matrix.
  ++ See \downlink{Manual Page}{manpageXXf07fdf}.
  f07fef : (String,Integer,Integer,Matrix DoubleFloat,_,
       Integer,Integer,Matrix DoubleFloat) -> Result
  ++ f07fef(uplo,n,nrhs,a,lda,ldb,b)
  ++ (DPOTRS) solves a real symmetric positive-definite system
  ++ of linear equations with multiple right-hand sides, AX=B, where A
  ++ has been factorized by F07FDF (DPOTRF).
  ++ See \downlink{Manual Page}{manpageXXf07fef}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(String)
import AnyFunctions1(Matrix Integer)

f07adf(mArg:Integer,nArg:Integer,ldaArg:Integer,_
aArg:Matrix DoubleFloat): Result ==
[(invokeNagman(NIL$Lisp,_
  "f07adf",_  
  ["ipiv":S,"info":S]$Lisp,_
  [["double":S,["a":S,"lda":S,"n":S]$Lisp_,
    "$ipiv":S,"info":S]$Lisp_ ,
  )\$Lisp_ ,
  ["integer":S,"m":S,"n":S,"lda":S,["ipiv":S,"m":S]$Lisp_ ,
    "info":S]$Lisp_ ,
  ]\$Lisp_ ,
  ["ipiv":S,"info":S,"a":S]$Lisp_ ,
  [(mArg::Any,nArg::Any,ldaArg::Any,aArg::Any ])_
  @List Any]$Lisp)$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))$Result

f07aef(transArg:String,nArg:Integer,nrhsArg:Integer,_
aArg:Matrix DoubleFloat,ldaArg:Integer,ipivArg:Matrix Integer,_
ldbArg:Integer,bArg:Matrix DoubleFloat): Result ==
[(invokeNagman(NIL$Lisp,_
  "f07aef",_  
  ["info":S]$Lisp_,
  [["double":S,["a":S,"lda":S,"n":S]$Lisp_ ,
    ["b":S,"ldb":S,"nrhs":S]$Lisp]$Lisp_,
  ,["integer":S,"nrhs":S,"lda":S,["ipiv":S,"n":S]$Lisp_ ,
    "ldb":S,"info":S]$Lisp_,
  ,["character":S,"trans":S]$Lisp_ ]\$Lisp_ ,
  ["info":S,"b":S]$Lisp_,
  [(transArg::Any,nArg::Any,nrhsArg::Any,ldbArg::Any_ 
    ,bArg::Any,aArg::Any,ipivArg::Any,bArg::Any ])_
  @List Any]$Lisp)$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))$Result

f07fdf(uploArg:String,nArg:Integer,ldaArg:Integer,_
aArg:Matrix DoubleFloat): Result ==
[(invokeNagman(NIL$Lisp,_
  "f07fd",_  
  ["info":S]$Lisp_,
  [["double":S,["a":S,"lda":S,"n":S]$Lisp_,
    "$ipiv":S,"info":S]$Lisp_ ,
  )\$Lisp_ ,
  ["integer":S,"n":S,"lda":S,"info":S]$Lisp_ ,
  ]\$Lisp_ ,
  ["ipiv":S,"info":S,"a":S]$Lisp_ ,
  [(uploArg::Any,nArg::Any,ldaArg::Any,aArg::Any ])_
  @List Any]$Lisp)$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))$Result
packagel NAGF01 NagMatrixOperationsPackage

pretend List (Record(key:Symbol,entry:Any))$Result

— NAGF07.dotabb —

"NAGF07" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF07"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGF07" \rightarrow "ALIST"

package NAGF01 NagMatrixOperationsPackage

— NagMatrixOperationsPackage.input —

)set break resume
)sys rm -f NagMatrixOperationsPackage.output
)spool NagMatrixOperationsPackage.output
This package uses the NAG Library to provide facilities for matrix factorizations and associated transformations.

F01 -- Matrix Factorization

Chapter F01
Matrix Factorization

1. Scope of the Chapter
This chapter provides facilities for matrix factorizations and associated transformations.

2. Background to the Problems
An n by n matrix may be factorized as

\[ T \]

\[ A=PLUQ \]

where L and U are respectively lower and upper triangular matrices, and P and Q are permutation matrices. This is called an LU factorization. For general dense matrices it is usual to choose Q=I and to then choose P to ensure that the factorization is numerically stable. For sparse matrices, judicious choice of P and Q ensures numerical stability as well as maintaining as much sparsity as possible in the factors L and U. The LU factorization is normally used in connection with the solution of the linear equations

\[ Ax=b, \]

whose solution, \( x \), may then be obtained by solving in succession the simpler equations
\[ T \quad Ly = Pb, \quad Uz = y, \quad x = Qz \]

the first by forward substitution and the second by backward substitution. Routines to perform this solution are to be found in Chapter F04.

When \( A \) is symmetric positive-definite then we can choose \( U = L \) and \( Q = P \), to give the Cholesky factorization. This factorization is numerically stable without permutations, but in the sparse case the permutations can again be used to try to maintain sparsity. The Cholesky factorization is sometimes expressed as

\[ T \quad A = PLDL^T P, \]

where \( D \) is a diagonal matrix with positive diagonal elements and \( L \) is unit lower triangular.

The LU factorization can also be performed on rectangular matrices, but in this case it is more usual to perform a QR factorization. When \( A \) is an \( m \) by \( n \) (\( m \geq n \)) matrix this is given by

\[
\begin{pmatrix}
R \\
A = Q^T(0),
\end{pmatrix}
\]

where \( R \) is an \( n \) by \( n \) upper triangular matrix and \( Q \) is an orthogonal (unitary in the complex case) matrix.

3. Recommendations on Choice and Use of Routines

Routine F07ADF performs the LU factorization of a real \( m \) by \( n \) dense matrix.

The LU factorization of a sparse matrix is performed by routine F01BRF. Following the use of F01BRF, matrices with the same sparsity pattern may be factorized by routine F01BSF.

The Cholesky factorization of a real symmetric positive-definite dense matrix is performed by routine F07FDF.

Routine F01MCF performs the Cholesky factorization of a real symmetric positive-definite variable band (skyline) matrix, and a general sparse symmetric positive-definite matrix may be factorized using routine F01MAF.

The QR factorization of an \( m \) by \( n \) (\( m \geq n \)) matrix is performed by
routine F01QCF in the real case, and F01RCF in the complex case. Following the use of F01QCF, operations with Q may be performed using routine F01QDF and some, or all, of the columns of Q may be formed using routine F01QEF. Routines F01RDF and F01REF perform the same tasks following the use of F01RCF.

F01 -- Matrix Factorizations

Chapter F01

Matrix Factorizations

F01BRF LU factorization of real sparse matrix

F01BSF LU factorization of real sparse matrix with known sparsity pattern

T

F01MAF LL factorization of real sparse symmetric positive-definite matrix

T

F01MCF LDL factorization of real symmetric positive-definite variable-bandwidth matrix

F01QCF QR factorization of real m by n matrix (m>=n)

T

F01QDF Operations with orthogonal matrices, compute QB or Q B after factorization by F01QCF

F01QEF Operations with orthogonal matrices, form columns of Q after factorization by F01QCF

F01RCF QR factorization of complex m by n matrix (m>=n)

H

F01RDF Operations with unitary matrices, compute QB or Q B after factorization by F01RCF

F01REF Operations with unitary matrices, form columns of Q after factorization by F01RCF

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations

F01BRF

F01BRF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01BRF factorizes a real sparse matrix. The routine either forms the LU factorization of a permutation of the entire matrix, or, optionally, first permutes the matrix to block lower triangular form and then only factorizes the diagonal blocks.

2. Specification

```fortran
SUBROUTINE F01BRF (N, NZ, A, LICN, IRN, LIRN, ICN, PIVOT, IKEEP, IW, W, LBLOCK, GROW, ABORT, IDISP, IFAIL)
  INTEGER N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN), IKEEP(5*N), IW(8*N), IDISP(10), IFAIL
  DOUBLE PRECISION A(LICN), PIVOT, W(N)
  LOGICAL LBLOCK, GROW, ABORT(4)
```

3. Description

Given a real sparse matrix A, this routine may be used to obtain the LU factorization of a permutation of A,

\[ PAQ = LU \]

where P and Q are permutation matrices, L is unit lower triangular and U is upper triangular. The routine uses a sparse variant of Gaussian elimination, and the pivotal strategy is designed to compromise between maintaining sparsity and controlling loss of accuracy through round-off.

Optionally the routine first permutes the matrix into block lower triangular form and then only factorizes the diagonal blocks. For some matrices this gives a considerable saving in storage and execution time.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve \( T \) sparse systems of linear equations \( Ax = b \) or \( A^T x = b \). If several matrices of the same sparsity pattern are to be factorized, F01BSF should be used for the second and subsequent matrices.

The method is fully described by Duff [1].

4. References
5. Parameters

1: N -- INTEGER  
   On entry: n, the order of the matrix A. Constraint: N > 0.

2: NZ -- INTEGER  
   On entry: the number of non-zero elements in the matrix A.  
   Constraint: NZ > 0.

3: A(LICN) -- DOUBLE PRECISION array  
   On entry: A(i), for i = 1,2,...,NZ must contain the non-zero elements of the sparse matrix A.  
   They can be in any order since the routine will reorder them. On exit: the non-zero elements in the LU factorization.  
   The array must not be changed by the user between a call of this routine and a call of F04AXF.

4: LICN -- INTEGER  
   On entry:  
   the dimension of the arrays A and ICN as declared in the (sub)program from which F01BRF is called.  
   Since the factorization is returned in A and ICN, LICN should be large enough to accommodate this and should ordinarily be 2 to 4 times as large as NZ.  
   Constraint: LICN >= NZ.

5: IRN(LIRN) -- INTEGER array  
   On entry: IRN(i), for i = 1,2,...,NZ must contain the row index of the non-zero element stored in A(i).  
   On exit: the array is overwritten and is not needed for subsequent calls of F01BSF or F04AXF.

6: LIRN -- INTEGER  
   On entry:  
   the dimension of the array IRN as declared in the (sub)program from which F01BRF is called.  
   It need not be as large as LICN; normally it will not need to be very much greater than NZ.  
   Constraint: LIRN >= NZ.

7: ICN(LICN) -- INTEGER array  
   On entry: ICN(i), for i = 1,2,...,NZ must contain the column index of the non-zero element stored in A(i).  
   On exit: the column indices of the non-zero elements in the factorization.  
   The array must not be changed by the user between a call of this routine and subsequent calls of
8: PIVOT -- DOUBLE PRECISION

On entry: PIVOT should have a value in the range 0.0 ≤
PIVOT ≤ 0.9999 and is used to control the choice of pivots.
If PIVOT < 0.0, the value 0.0 is assumed, and if PIVOT > 0.
9999, the value 0.9999 is assumed. When searching a row for
a pivot, any element is excluded which is less than PIVOT
times the largest of those elements in the row available as
pivots. Thus decreasing PIVOT biases the algorithm to
maintaining sparsity at the expense of stability. Suggested
value: PIVOT = 0.1 has been found to work well on test
examples.

9: IKEEP(5*N) -- INTEGER array

On exit: indexing information about the factorization. The
array must not be changed by the user between a call of this
routine and calls of F01BSF or F04AXF.

10: IW(8*N) -- INTEGER array

Workspace

11: W(N) -- DOUBLE PRECISION array

On exit: if GROW = .TRUE., W(1) contains an estimate (an
upper bound) of the increase in size of elements encountered
during the factorization (see GROW); the rest of the array
is used as workspace.

If GROW = .FALSE., the array is not used.

12: LBLOCK -- LOGICAL

On entry: if LBLOCK = .TRUE., the matrix is pre-ordered
into block lower triangular form before the LU factorization
is performed; otherwise the entire matrix is factorized.
Suggested value: LBLOCK = .TRUE. unless the matrix is known
to be irreducible.

13: GROW -- LOGICAL

On entry: if GROW = .TRUE., then on exit W(1) contains an
estimate (an upper bound) of the increase in size of
elements encountered during the factorization. If the matrix
is well-scaled (see Section 8.2), then a high value for W(1)
indicates that the LU factorization may be inaccurate and
the user should be wary of the results and perhaps increase
the parameter PIVOT for subsequent runs (see Section 7).
Suggested value: GROW = .TRUE..

14: ABORT(4) -- LOGICAL array

On entry:

if ABORT(1) = .TRUE., the routine will exit
immediately on detecting a structural singularity (one
that depends on the pattern of non-zeros) and return

\( IFAIL = 1 \); otherwise it will complete the
factorization (see Section 8.3).

If \( ABORT(2) = .TRUE. \), the routine will exit
immediately on detecting a numerical singularity (one
that depends on the numerical values) and return \( IFAIL = 2 \); otherwise it will complete the factorization (see
Section 8.3).

If \( ABORT(3) = .TRUE. \), the routine will exit
immediately (with \( IFAIL = 5 \)) when the arrays \( A \) and \( ICN \)
are filled up by the previously factorized, active and
unfactorized parts of the matrix; otherwise it continues so that better guidance on necessary array
sizes can be given in \( IDISP(6) \) and \( IDISP(7) \), and will
exit with \( IFAIL \) in the range 4 to 6. Note that there
is always an immediate error exit if the array \( IRN \) is
too small.

If \( ABORT(4) = .TRUE. \), the routine exits immediately
(with \( IFAIL = 13 \)) if it finds duplicate elements in
the input matrix. If \( ABORT(4) = .FALSE. \), the routine
proceeds using a value equal to the sum of the
duplicate elements. In either case details of each
duplicate element are output on the current advisory
message unit (see \( X04ABF \)), unless suppressed by the
value of \( IFAIL \) on entry.

Suggested values:
\( ABORT(1) = .TRUE. \)
\( ABORT(2) = .TRUE. \)
\( ABORT(3) = .FALSE. \)
\( ABORT(4) = .TRUE. \)

15: \( IDISP(10) \) -- INTEGER array
Output
On exit: \( IDISP \) is used to communicate information about the
factorization to the user and also between a call of \( F01BRF \)
and subsequent calls to \( F01BSF \) or \( F04AXF \).

\( IDISP(1) \) and \( IDISP(2) \), indicate the position in arrays
\( A \) and \( ICN \) of the first and last elements in the LU
factorization of the diagonal blocks. (\( IDISP(2) \) gives
the number of non-zeros in the factorization.)

\( IDISP(3) \) and \( IDISP(4) \), monitor the adequacy of 'elbow
room' in the arrays \( IRN \) and \( A/ICN \) respectively, by
giving the number of times that the data in these
arrays has been compressed during the factorization to release more storage. If either IDISP(3) or IDISP(4) is quite large (say greater than 10), it will probably pay the user to increase the size of the corresponding array(s) for subsequent runs. If either is very low or zero, then the user can perhaps save storage by reducing the size of the corresponding array(s).

IDISP(5), gives an upper bound on the rank of the matrix.

IDISP(6) and IDISP(7), give the minimum size of arrays IRN and A/ICN respectively which would enable a successful run on an identical matrix (but some 'elbow-room' should be allowed - see Section 8).

IDISP(8) to (10), are only used if LBLOCK = .TRUE..

IDISP(8), gives the structural rank of the matrix.

IDISP(9), gives the number of diagonal blocks.

IDISP(10), gives the size of the largest diagonal block.

IDISP(1) and IDISP(2), must not be changed by the user between a call of F01BRF and subsequent calls to F01BSF or F04AXF.

16: IFAIL -- INTEGER Input/Output

For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, IFAIL must be set to a value with the decimal expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.

a=0  specifies hard failure, otherwise soft failure;

b=0  suppresses error messages, otherwise error messages will be printed (see Section 6);

b=0  suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.
6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL=-2
Successful factorization of a numerically singular matrix (which may also be structurally singular) (see Section 8.3).

IFAIL=-1
Successful factorization of a structurally singular matrix (see Section 8.3).

IFAIL= 1
The matrix is structurally singular and the factorization has been abandoned (ABORT(1) was .TRUE. on entry).

IFAIL= 2
The matrix is numerically singular and the factorization has been abandoned (ABORT(2) was .TRUE. on entry).

IFAIL= 3
LIRN is too small: there is not enough space in the array IRN to continue the factorization. The user is recommended to try again with LIRN (and the length of IRN) equal to at least IDISP(6) + N/2.

IFAIL= 4
LICN is much too small: there is much too little space in the arrays A and ICN to continue the factorization.

IFAIL= 5
LICN is too small: there is not enough space in the arrays A and ICN to store the factorization. If ABORT(3) was .FALSE. on entry, the factorization has been completed but some of the LU factors have been discarded to create space, IDISP(7) then gives the minimum value of LICN (i.e., the minimum length of A and ICN) required for a successful factorization of the same matrix.

IFAIL= 6
LICN and LIRN are both too small: effectively this is a combination of IFAIL = 3 and IFAIL = 5 (with ABORT(3) = .FALSE.).

IFAIL= 7
LICN is too small: there is not enough space in the arrays A and ICN for the permutation to block triangular form.

IFAIL= 8
On entry N <= 0.

IFAIL= 9
On entry NZ <= 0.

IFAIL= 10
On entry LICN < NZ.

IFAIL= 11
On entry LIRN < NZ.

IFAIL= 12
On entry an element of the input matrix has a row or column index (i.e., an element of IRN or ICN) outside the range 1 to N.

IFAIL= 13
Duplicate elements have been found in the input matrix and the factorization has been abandoned (ABORT(4) = .TRUE. on entry).

7. Accuracy

The factorization obtained is exact for a perturbed matrix whose (i,j)th element differs from a by less than 3(epsilon)(rho)m
where (epsilon) is the machine precision, (rho) is the growth value returned in W(1) if GROW = .TRUE., and m the number of
Gaussian elimination operations applied to element (i,j). The value of m is not greater than n and is usually much less.
Small (rho) values therefore guarantee accurate results, but unfortunately large (rho) values may give a very pessimistic indication of accuracy.

8. Further Comments

8.1. Timing

The time required may be estimated very roughly from the number (tau) of non-zeros in the factorized form (output as IDISP(2)) and for this routine and its associates is

\[ \text{F01BRF: } 5(\tau)/n \text{ units} \]
2
F01BSF: (tau) /n units

F04AXF: 2(tau) units

where our unit is the time for the inner loop of a full matrix code (e.g. solving a full set of equations takes about \( -n^{3/3} \) units). Note that the faster F01BSF time makes it well worthwhile to use this for a sequence of problems with the same pattern.

It should be appreciated that (tau) varies widely from problem to problem. For network problems it may be little greater than NZ, the number of non-zeros in A; for discretisation of 2-dimensional and 3-dimensional partial differential equations it may be about \( 1.5^{3/3} \) \( 3\log n \) and \( -n^{2/2} \), respectively.

The time taken to find the block lower triangular form (LBLOCK = it is not found (LBLOCK = .FALSE.). If the matrix is irreducible (IDISP(9) = 1 after a call with LBLOCK = .TRUE.) then this time is wasted. Otherwise, particularly if the largest block is small (IDISP(10)<<n), the consequent savings are likely to be greater.

The time taken to estimate growth (GROW = .TRUE.) is typically under 2% of the overall time.

The overall time may be substantially increased if there is inadequate 'elbow-room' in the arrays A, IRN and ICN. When the sizes of the arrays are minimal (IDISP(6) and IDISP(7)) it can execute as much as three times slower. Values of IDISP(3) and IDISP(4) greater than about 10 indicate that it may be worthwhile to increase array sizes.

8.2. Scaling

The use of a relative pivot tolerance PIVOT essentially presupposes that the matrix is well-scaled, i.e., that the matrix elements are broadly comparable in size. Practical problems are often naturally well-scaled but particular care is needed for problems containing mixed types of variables (for example millimetres and neutron fluxes).

8.3. Singular and Rectangular Systems

It is envisaged that this routine will almost always be called for square non-singular matrices and that singularity indicates
an error condition. However, even if the matrix is singular it is possible to complete the factorization. It is even possible for F04AXF to solve a set of equations whose matrix is singular provided the set is consistent.

Two forms of singularity are possible. If the matrix would be singular for any values of the non-zeros (e.g. if it has a whole row of zeros), then we say it is structurally singular, and continue only if ABORT(1) = .FALSE.. If the matrix is non-singular by virtue of the particular values of the non-zeros, then we say that it is numerically singular and continue only if ABORT(2) = .FALSE..

Rectangular matrices may be treated by setting N to the larger of the number of rows and numbers of columns and setting ABORT(1) =

Note: the soft failure option should be used (last digit of IFAIL = 1) if the user wishes to factorize singular matrices with ABORT (1) or ABORT(2) set to .FALSE..

8.4. Duplicated Non-zeros

The matrix A may consist of a sum of contributions from different sub-systems (for example finite elements). In such cases the user may rely on this routine to perform assembly, since duplicated elements are summed.

9. Example

To factorize the real sparse matrix:

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 & 0 \\
-2 & 0 & 0 & 1 & 1 & 0. \\
-1 & 0 & 0 & -1 & 2 & -3 \\
-1 & -1 & 0 & 0 & 0 & 6
\end{pmatrix}
\]

This example program simply prints out some information about the factorization as returned by F01BRF in W(1) and IDISP. Normally the call of F01BRF would be followed by a call of F04AXF (see Example for F04AXF).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
CHAPTER 15. CHAPTER N

F01BSF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01BSF factorizes a real sparse matrix using the pivotal sequence previously obtained by F01BRF when a matrix of the same sparsity pattern was factorized.

2. Specification

```fortran
SUBROUTINE F01BSF (N, NZ, A, LICN, IVECT, JVECT, ICN,
1     IKEEP, IW, GROW, ETA, RPMIN, ABORT,
2     IDISP, IFAIL)
  INTEGER N, NZ, LICN, IVECT(NZ), JVECT(NZ), ICN
1     (LICN), IKEEP(5*N), IW(8*N), IDISP(2),
2     IFAIL
  DOUBLE PRECISION A(LICN), W(N), ETA, RPMIN
  LOGICAL GROW, ABORT
```

3. Description

This routine accepts as input a real sparse matrix of the same sparsity pattern as a matrix previously factorized by a call of F01BRF. It first applies to the matrix the same permutations as were used by F01BRF, both for permutation to block triangular form and for pivoting, and then performs Gaussian elimination to obtain the LU factorization of the diagonal blocks.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve $T$ sparse systems of linear equations $Ax=b$ or $A^x=b$.

F01BSF is much faster than F01BRF and in some applications it is expected that there will be many calls of F01BSF for each call of F01BRF.

The method is fully described in Duff [1].

4. References

5. Parameters

1: N -- INTEGER
   On entry: n, the order of the matrix A. Constraint: N > 0.

2: NZ -- INTEGER
   On entry: the number of non-zeros in the matrix A.
   Constraint: NZ > 0.

3: A(LICN) -- DOUBLE PRECISION array
   On entry: A(i), for i = 1,2,...,NZ must contain the non-zero elements of the sparse matrix A. They can be in any order since the routine will reorder them. On exit: the non-zero elements in the factorization. The array must not be changed by the user between a call of this routine and a call of F04AXF.

4: LICN -- INTEGER
   On entry: the dimension of the arrays A and ICN as declared in the (sub)program from which F01BSF is called.
   It should have the same value as it had for F01BRF.
   Constraint: LICN >= NZ.

5: IVECT(NZ) -- INTEGER array

6: JVECT(NZ) -- INTEGER array
   On entry: IVECT(i) and JVECT(i), for i = 1,2,...,NZ must contain the row index and the column index respectively of the non-zero element stored in A(i).

7: ICN(LICN) -- INTEGER array
   On entry: the same information as output by F01BRF. It must not be changed by the user between a call of this routine and a call of F04AXF.

8: IKEEP(5*N) -- INTEGER array
   On entry: the same indexing information about the factorization as output from F01BRF. It must not be changed between a call of this routine and a call of F04AXF.

9: IW(8*N) -- INTEGER array
   Workspace

10: W(N) -- DOUBLE PRECISION array
    On exit: if GROW = .TRUE., W(1) contains an estimate (an upper bound) of the increase in size of elements encountered during the factorization (see GROW); the rest of the array is used as workspace.
If GROW = .FALSE., the array is not used.

11: GROW -- LOGICAL
   Input
   On entry: if GROW = .TRUE., then on exit W(1) contains an
   estimate (an upper bound) of the increase in size of
   elements encountered during the factorization. If the matrix
   is well-scaled (see Section 8.2), then a high value for W(1)
   indicates that the LU factorization may be inaccurate and
   the user should be wary of the results and perhaps increase
   the parameter PIVOT for subsequent runs (see Section 7).

12: ETA -- DOUBLE PRECISION
   Input
   On entry: the relative pivot threshold below which an error
   diagnostic is provoked and IFAIL is set to 7. If ETA is
   greater than 1.0, then no check on pivot size is made.
   -4
   Suggested value: ETA = 10.

13: RPMIN -- DOUBLE PRECISION
    Output
    On exit: if ETA is less than 1.0, then RPMIN gives the
    smallest ratio of the pivot to the largest element in the
    row of the corresponding upper triangular factor thus
    monitoring the stability of the factorization. If RPMIN is
    very small it may be advisable to perform a new
    factorization using F01BRF.

14: ABORT -- LOGICAL
    Input
    On entry: if ABORT = .TRUE., the routine exits immediately
    (with IFAIL = 8) if it finds duplicate elements in the input
    matrix. If ABORT = .FALSE., the routine proceeds using a
    value equal to the sum of the duplicate elements. In either
    case details of each duplicate element are output on the
    current advisory message unit (see X04ABF), unless
    suppressed by the value of IFAIL on entry. Suggested value:
    ABORT = .TRUE..

15: IDISP(2) -- INTEGER array
    Input
    On entry: IDISP(1) and IDISP(2) must be unchanged since the
    previous call of F01BRF.

16: IFAIL -- INTEGER
    Input/Output
    For this routine, the normal use of IFAIL is extended to
    control the printing of error and warning messages as well
    as specifying hard or soft failure (see the Essential
    Introduction).

    Before entry, IFAIL must be set to a value with the decimal
    expansion cba, where each of the decimal digits c, b and a
    must have a value of 0 or 1.
a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL= 1
On entry N <= 0.

IFAIL= 2
On entry NZ <= 0.

IFAIL= 3
On entry LICN < NZ.

IFAIL= 4
On entry an element of the input matrix has a row or column index (i.e., an element of IVECT or JVECT) outside the range 1 to N.

IFAIL= 5
The input matrix is incompatible with the matrix factorized by the previous call of F01BRF (see Section 8).

IFAIL= 6
The input matrix is numerically singular.

IFAIL= 7
A very small pivot has been detected (see Section 5, ETA). The factorization has been completed but is potentially unstable.

IFAIL= 8
Duplicate elements have been found in the input matrix and the factorization has been abandoned (ABORT = .TRUE. on
7. Accuracy

The factorization obtained is exact for a perturbed matrix whose $(i,j)$th element differs from $a$ by less than $3(\varepsilon)(\rho)m_{ij}$ where $\varepsilon$ is the machine precision, $\rho$ is the growth value returned in $W(1)$ if $GROW = .TRUE.$, and $m$ the number of Gaussian elimination operations applied to element $(i,j)$.

If $\rho = W(1)$ is very large or $RPMIN$ is very small, then a fresh call of F01BRF is recommended.

8. Further Comments

If the user has a sequence of problems with the same sparsity pattern then this routine is recommended after F01BRF has been called for one such problem. It is typically 4 to 7 times faster but is potentially unstable since the previous pivotal sequence is used. Further details on timing are given in document F01BRF.

If growth estimation is performed ($GROW = .TRUE.$), then the time increases by between 5% and 10%. Pivot size monitoring ($\eta \leq 1.0$) involves a similar overhead.

We normally expect this routine to be entered with a matrix having the same pattern of non-zeros as was earlier presented to F01BRF. However there is no record of this pattern, but rather a record of the pattern including all fill-ins. Therefore we permit additional non-zeros in positions corresponding to fill-ins.

If singular matrices are being treated then it is also required that the present matrix be sufficiently like the previous one for the same permutations to be suitable for factorization with the same set of zero pivots.

9. Example

To factorize the real sparse matrices

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 \\
0 & 0 & 3 & 0 & 0 \\
-2 & 0 & 0 & 1 & 1 \\
-1 & 0 & 0 & -1 & 2 \\
-1 & -1 & 0 & 0 & 6
\end{pmatrix}
\]

and
This example program simply prints the values of W(1) and RPMIN returned by F01BSF. Normally the calls of F01BRF and F01BSF would be followed by calls of F04AXF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
diagonal matrix with positive diagonal elements, \( E \) is an error matrix representing elements dropped during the factorization and diagonal elements that have been modified to ensure that \( C \) is positive-definite, and \( W \) is a diagonal matrix, chosen to make the diagonal elements of \( WAW \) unity.

\[-1 -1\]

\( W \) \( CW \) is a pre-conditioning matrix for \( A \), and the factorization of \( C \) is intended to be used by F04MAF to solve systems of linear equations \( Ax = b \).

The permutation matrix \( P \) is chosen to reduce the amount of fill-in that occurs in \( L \) and the user-supplied parameter DROPTL can also be used to control the amount of fill-in that occurs.

Full details on the factorization can be found in Munksgaard [1].

F01MAF is based on the Harwell Library routine MA31A.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
   On entry: \( n \), the order of the matrix \( A \). Constraint: \( N \geq 1 \).

2: \( NZ \) -- INTEGER Input
   On entry: the number of non-zero elements in the upper triangular part of the matrix \( A \), including the number of elements on the leading diagonal. Constraint: \( NZ \geq N \).

3: \( A(LICN) \) -- DOUBLE PRECISION array Input/Output
   On entry: the first \( NZ \) elements of the array \( A \) must contain the non-zero elements of the upper triangular part of the sparse positive-definite symmetric matrix \( A \), including the elements on the leading diagonal. On exit: the first \( (NZ-N) \) elements of \( A \) contain the elements above the diagonal of the matrix \( WAW \), where \( W \) is a diagonal matrix whose \( i \)th diagonal element is \( w = a \). These elements are returned in order by rows and the value returned in \( ICN(k) \) gives the column index of the element returned in \( A(k) \). The value \( w \) is returned in the \( i \)th element of the array \( WKEEP \). The remaining \( LROW-NZ+N \) elements of \( A \), where \( LROW \) is the value returned in \( INFORM(1) \),
return details of the factorization for use by F04MAF.

4: LICN -- INTEGER 
   Input
   On entry:
   the dimension of the array A as declared in the (sub)program from which F01MAF is called.
   If fill-in is expected during the factorization, then a larger value of LICN will allow fewer elements to be dropped during the factorization, thus giving a more accurate factorization, which in turn will almost certainly mean that fewer iterations will be required by F04MAF. Constraint: LICN > = 2*NZ.

5: IRN(LIRN) -- INTEGER array 
   Input/Output
   On entry: IRN(k), for k = 1,2,...,NZ must contain the row index of the non-zero element of the matrix A supplied in A(k). On exit: the first LCOL elements of IRN, where LCOL is the value returned in INFORM(2), return details of the factorization for use by F04MAF.

6: LIRN -- INTEGER 
   Input
   On entry:
   the dimension of the array IRN as declared in the (sub)program from which F01MAF is called.
   LIRN must be at least NZ, but, as with LICN, if fill-in is expected then a larger value of LIRN will allow a more accurate factorization. For this purpose LIRN should exceed NZ by the same amount that LICN exceeds 2*NZ. Constraint: LIRN >= NZ.

7: ICN(LICN) -- INTEGER array 
   Input/Output
   On entry: ICN(k), for k = 1,2,...,NZ must contain the column index of the non-zero element of the matrix A supplied in A(k). Thus a \( = A(k) \), where \( i = IRN(k) \) and \( j = ICN(k) \). On exit: the first (NZ-N) elements of ICN give the column indices of the first (NZ-N) elements returned in A. The remaining LROW - NZ + N elements of ICN return details of the factorization for use by F04MAF.

8: DROPTL -- DOUBLE PRECISION 
   Input/Output
   On entry: a value in the range \([-1.0, 1.0]\) to be used as a tolerance in deciding whether or not to drop elements during the factorization. At the kth pivot step the element a\(_{ij}\) is dropped if it would cause fill-in and if

\[
\frac{(k+1)}{(k)} \leq DROPTL
\]
If DROPTL is supplied as negative, then it is not altered during the factorization and so is unchanged on exit, but if DROPTL is supplied as positive then it may be altered by the routine with the aim of obtaining an accurate factorization in the space available. If DROPTL is supplied as -1.0, then no fill-in will occur during the factorization; and if DROPTL is supplied as 0.0 then a complete factorization is performed. On exit: may be overwritten with the value used by the routine in order to obtain an accurate factorization in the space available, if DROPTL > 0.0 on entry.

9: DENSW -- DOUBLE PRECISION Input/Output
On entry: a value in the range [0.0,1.0] to be used in deciding whether or not to regard the active part of the matrix at the kth pivot step as being full. If the ratio of non-zero elements to the total number of elements is greater than or equal to DENSW, then the active part is regarded as full. If DENSW < 1.0, then the storage used is likely to increase compared to the case where DENSW = 0, but the execution time is likely to decrease. Suggested value: DENSW = 0.8. On exit: if on entry DENSW is not in the range [0.0,1.0], then it is set to 0.8. Otherwise it is unchanged.

10: WKEEP(3*N) -- DOUBLE PRECISION array Output
On exit: information which must be passed unchanged to F04MAF. The first N elements contain the values w_i, for i = 1,2,...,n, and the next N elements contain the diagonal elements of D.

11: IKEEP(2*N) -- INTEGER array Output
On exit: information which must be passed unchanged to F04MAF.

12: IWORK(6*N) -- INTEGER array Workspace

13: ABORT(3) -- LOGICAL array Input
On entry:
if ABORT(1) = .TRUE., the routine will exit immediately on detecting duplicate elements and return IFAIL = 5. Otherwise when ABORT(1) = .FALSE., the calculations will continue using the sum of the duplicate entries. In either case details of the duplicate elements are output on the current advisory message unit (see X04ABF), unless suppressed by the value of IFAIL on entry.

If ABORT(2) = .TRUE., the routine will exit
immediately on detecting a zero or negative pivot element and return IFAIL = 6. Otherwise when ABORT(2) = .FALSE., the zero or negative pivot element will be modified to ensure positive-definiteness and a message will be printed on the current advisory message unit, unless suppressed by the value of IFAIL on entry.

If ABORT(3) = .TRUE., the routine will exit immediately if the arrays A and ICN have been filled up and return IFAIL = 7. Otherwise when ABORT(3) = .FALSE., the data in the arrays is compressed to release more storage and a message will be printed on the current advisory message unit, unless suppressed by the value of IFAIL on entry. If DROPTL is positive on entry, it may be modified in order to allow a factorization to be completed in the available space.

Suggested values:
- ABORT(1) = .TRUE.,
- ABORT(2) = .TRUE.,
- ABORT(3) = .TRUE..

14: INFORM(4) -- INTEGER array

On exit:
INFORM(1) returns the number of elements of A and ICN that have been used by the routine. Thus at least the first INFORM(1) elements of A and of ICN must be supplied to F04MAF.

Similarly, INFORM(2) returns the number of elements of IRN that have been used by the routine and so at least the first INFORM(2) elements must be supplied to F04MAF.

INFORM(3) returns the number of entries supplied in A that corresponded to diagonal and duplicate elements. If no duplicate entries were found, then INFORM(3) will return the value of N.

INFORM(4) returns the value k of the pivot step from which the active matrix was regarded as full.
INFORM must be passed unchanged to F04MAF.

15: IFAIL -- INTEGER

For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).
Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a
must have a value of 0 or 1.
a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages
will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning
messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e.,
hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL
contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
On entry N < 1,
or NZ < N,
or LIRN < NZ,
or LICN<2*NZ.

IFAIL= 2
One of the conditions 0 < IRN(k) <= ICN(k) <= N is not
satisfied so that A(k) is not in the upper triangle of the
matrix. No further computation is attempted.

IFAIL= 3
One of the diagonal elements of the matrix A is zero or
negative so that A is not positive-definite. No further
computation is attempted.

IFAIL= 4
The available space has been used and no further
compressions are possible. The user should either increase
DROPTL, or allocate more space to A, IRN and ICN.

For all the remaining values of IFAIL the computations will
continue in the case of soft failure, so that more than one
advisory message may be printed.

IFAIL= 5
Duplicate elements have been detected and ABORT(1) = .TRUE..

IFAIL= 6
A zero or negative pivot element has been detected during the factorization and ABORT(2) = .TRUE..

This should not happen if A is an M-matrix (see Munksgaard [1]), but may occur for other types of positive-definite matrix.

IFAIL= 7
The available space has been used and ABORT(3) = .TRUE..

7. Accuracy

The accuracy of the factorization will be determined by the size of the elements that are dropped and the size of the modifications made to the diagonal elements. If these sizes are small then the computed factors will correspond to a matrix close to A and the number of iterations required by F04MAF will be small. The more incomplete the factorization, the higher the number of iterations required by F04MAF.

8. Further Comments

The time taken by the routine will depend upon the sparsity pattern of the matrix and the number of fill-ins that occur during the factorization. At the very least the time taken can be expected to be roughly proportional to n(tau), where (tau) is the number of non-zeros.

The routine is intended for use with positive-definite matrices, but the user is warned that it will not necessarily detect non-positive-definiteness. Indeed the routine may return a factorization that can satisfactorily be used by F04MAF even when A is not positive-definite, but this should not be relied upon as F04MAF may not converge.

9. Example

The example program illustrates the use of F01MAF in conjunction with F04MAF to solve the 16 linear equations Ax=b, where

\[
\begin{pmatrix}
1 & z & z \\
z & 1 & z \\
z & 1 & z \\
z & 0 & 1 & z \\
z & 0 & 1 & z \\
\end{pmatrix}
\]
The \( n \) by \( n \) matrix \( A \) arises in the solution of Laplace's equation in a unit square, using a 5-point formula with a 6 by 6 discretisation, with unity on the boundaries.

The drop tolerance, \( \text{DROPTL} \), is taken as 0.1, and the density factor, \( \text{DENSW} \), is taken as 0.8. The value \( \text{IFAIL} = 111 \) is used so that advisory and error messages will be printed, but soft failure would occur if \( \text{IFAIL} \) were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F01MCF computes the Cholesky factorization of a real symmetric positive-definite variable-bandwidth matrix.

2. Specification

SUBROUTINE F01MCF (N, A, LAL, NROW, AL, D, IFAIL)
INTEGER N, LAL, NROW(N), IFAIL
DOUBLE PRECISION A(LAL), AL(LAL), D(N)

3. Description

This routine determines the unit lower triangular matrix L and
\[ T \]
the diagonal matrix D in the Cholesky factorization \( A=LDL \) of a symmetric positive-definite variable-bandwidth matrix A of order n. (Such a matrix is sometimes called a 'sky-line' matrix.)

The matrix A is represented by the elements lying within the envelope of its lower triangular part, that is, between the first non-zero of each row and the diagonal (see Section 9 for an example). The width NROW(i) of the ith row is the number of elements between the first non-zero element and the element on the diagonal, inclusive. Although, of course, any matrix possesses an envelope as defined, this routine is primarily intended for the factorization of symmetric positive-definite matrices with an average bandwidth which is small compared with n (also see Section 8).

The method is based on the property that during Cholesky factorization there is no fill-in outside the envelope.

The determination of L and D is normally the first of two steps in the solution of the system of equations \( Ax=b \). The remaining step, viz. the solution of \( LDL x=b \), may be carried out using F04MCF.

4. References


5. Parameters

1: N -- INTEGER
Input
On entry: n, the order of the matrix A. Constraint: N >= 1.

2: A(LAL) -- DOUBLE PRECISION array
   On entry: the elements within the envelope of the lower triangle of the positive-definite symmetric matrix A, taken in row by row order. The following code assigns the matrix elements within the envelope to the correct elements of the array:
   
   K = 0
   DO 20 I = 1, N
   DO 10 J = I-NROW(I)+1, I
   K = K + 1
   A(K) = matrix (I,J)
   10 CONTINUE
   20 CONTINUE

   See also Section 8.

3: LAL -- INTEGER
   On entry: the smaller of the dimensions of the arrays A and AL as declared in the calling (sub)program from which F01MCF is called. Constraint: LAL >= NROW(1) + NROW(2) +... + NROW(n).

4: NROW(N) -- INTEGER array
   On entry: NROW(i) must contain the width of row i of the matrix A, i.e., the number of elements between the first (leftmost) non-zero element and the element on the diagonal, inclusive. Constraint: 1 <= NROW(i) <= i, for i=1,2,...,n.

5: AL(LAL) -- DOUBLE PRECISION array
   On exit: the elements within the envelope of the lower triangular matrix L, taken in row by row order. The envelope of L is identical to that of the lower triangle of A. The unit diagonal elements of L are stored explicitly. See also Section 8.

6: D(N) -- DOUBLE PRECISION array
   On exit: the diagonal elements of the the diagonal matrix D. Note that the determinant of A is equal to the product of these diagonal elements. If the value of the determinant is required it should not be determined by forming the product explicitly, because of the possibility of overflow or underflow. The logarithm of the determinant may safely be formed from the sum of the logarithms of the diagonal elements.

7: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry N < 1,

or for some i, NROW(i) < 1 or NROW(i) > i,

or LAL < NROW(1) + NROW(2) +... + NROW(N).

IFAIL= 2
A is not positive-definite, or this property has been destroyed by rounding errors. The factorization has not been completed.

IFAIL= 3
A is not positive-definite, or this property has been destroyed by rounding errors. The factorization has been completed but may be very inaccurate (see Section 7).

7. Accuracy

If IFAIL = 0 on exit, then the computed L and D satisfy the relation LDL = A+F, where

$$
||F|| \leq km (\text{epsilon}) \max_{i} a_{ii}
$$

and

$$
||F|| \leq km (\text{epsilon}) ||A||
$$

where k is a constant of order unity, m is the largest value of NROW(i), and (epsilon) is the machine precision. See Wilkinson and Reinsch [2], pp 25--27, 54--55. If IFAIL = 3 on exit, then the factorization has been completed although the matrix was not positive-definite. However the factorization may be very inaccurate and should be used only with great caution. For instance, if it is used to solve a set of equations Ax=b using
F04MCF, the residual vector $b-Ax$ should be checked.

8. Further Comments

The time taken by the routine is approximately proportional to the sum of squares of the values of NROW(i).

The distribution of row widths may be very non-uniform without undue loss of efficiency. Moreover, the routine has been designed to be as competitive as possible in speed with routines designed for full or uniformly banded matrices, when applied to such matrices.

Unless otherwise stated in the Users’ Note for your implementation, the routine may be called with the same actual array supplied for parameters A and AL, in which case L overwrites the lower triangle of A. However this is not standard Fortran 77 and may not work in all implementations.

9. Example

To obtain the Cholesky factorization of the symmetric matrix, whose lower triangle is:

\[
\begin{pmatrix}
1 & & & & \\
2 & 5 & & & \\
0 & 3 & 13 & & \\
0 & 0 & 0 & 16 & \\
5 & 14 & 18 & 8 & 55 \\
0 & 0 & 0 & 24 & 17 & 77
\end{pmatrix}
\]

For this matrix, the elements of NROW must be set to 1, 2, 2, 1, 5, 3, and the elements within the envelope must be supplied in row order as:

1, 2, 5, 3, 13, 16, 5, 14, 18, 8, 55, 24, 17, 77.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
not included in the Foundation Library.

1. Purpose

F01QCF finds the QR factorization of the real m by n matrix A, where m>=n.

2. Specification

SUBROUTINE F01QCF (M, N, A, LDA, ZETA, IFAIL)
INTEGER M, N, LDA, IFAIL
DOUBLE PRECISION A(LDA,*), ZETA(*)

3. Description

The m by n matrix A is factorized as

\[
\begin{align*}
A &= Q(0) & \text{when } m > n, \\
A &= QR & \text{when } m = n,
\end{align*}
\]

where Q is an m by m orthogonal matrix and R is an n by n upper triangular matrix. The factorization is obtained by Householder’s method. The kth transformation matrix, Q, which is used to introduce zeros into the kth column of A is given in the form

\[
Q_k = \begin{pmatrix}
I & 0 \\
0 & T_k
\end{pmatrix}
\]

where

\[
T_k = I - uu^T, \quad k = 1, 2, \ldots, n
\]

\[
\begin{pmatrix}
(zeta) \\
\vdots \\
0
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 \\
0
\end{pmatrix}, \quad k = 1, 2, \ldots, n
\]

(zeta) is a scalar and z is an (m-k) element vector. (zeta)

and z are chosen to annihilate the elements below the triangular k
part of A.

The vector u is returned in the kth element of the array ZETA
and in the kth column of A, such that (zeta) is in ZETA(k) and 
the elements of z are in A(k+1,k),...,A(m,k). The elements of R 
are returned in the upper triangular part of A.

Q is given by

\[ Q^T = (Q_0 Q_1 \ldots Q_{n-1}) \]

Good background descriptions to the QR factorization are given in 
Dongarra et al [1] and Golub and Van Loan [2], but note that this 
routine is not based upon LINPACK routine DQRDC.

4. References

LINPACK Users' Guide. SIAM, Philadelphia.

Edition). Johns Hopkins University Press, Baltimore, 
Maryland.

Oxford University Press.

5. Parameters

1: M -- INTEGER Input 
   On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER Input 
   On entry: n, the number of columns of A.

   When N = 0 then an immediate return is effected. 
   Constraint: N >= 0.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output 
   Note: the second dimension of the array A must be at least 
   max(1,n).
   On entry: the leading m by n part of the array A must 
   contain the matrix to be factorized. On exit: the n by n 
   upper triangular part of A will contain the upper triangular 
   matrix R and the m by n strictly lower triangular part of A 
   will contain details of the factorization as described in 
   Section 3.
4: LDA -- INTEGER   Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which F01QCF is called.
   Constraint: LDA >= max(1,M).

5: ZETA(*) -- DOUBLE PRECISION array   Output
   Note: the dimension of the array ZETA must be at least max
   (1,n) On exit: ZETA(k) contains the scalar (zeta) for the k
   th transformation. If T =I then ZETA(k)=0.0, otherwise ZETA( k
   k) contains (zeta) as described in Section 3 and (zeta) is
   k
   always in the range (1.0, \sqrt{2.0}).

6: IFAIL -- INTEGER   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AA).

IFAIL=-1
   On entry M < N,
   or    N < 0,
   or    LDA < M.

7. Accuracy

The computed factors Q and R satisfy the relation

\[(R)\]
\[Q(0)=A+E,\]

where

\[|||E|||<=c(\text{epsilon})|||A|||,\]
and (epsilon) is the machine precision (see X02AJF(*)), c is a modest function of m and n and \( \| \cdot \| \) denotes the spectral (two) norm.

8. Further Comments

The approximate number of floating-point operations is given by:

\[
2n (3m-n)/3.
\]

Following the use of this routine the operations

\[
T
\]

\[
B:=QB \quad \text{and} \quad B:=Q^T B,
\]

where B is an m by k matrix, can be performed by calls to F01QDF. The operation B:=QB can be obtained by the call:

\[
\begin{align*}
&\text{IFAIL} = 0 \\
&\text{CALL F01QDF('No transpose', 'Separate', M, N, A, LDA, ZETA, K, B, LDB, WORK, IFAIL)}
\end{align*}
\]

and B:=Q B can be obtained by the call:

\[
\begin{align*}
&\text{IFAIL} = 0 \\
&\text{CALL F01QDF('Transpose', 'Separate', M, N, A, LDA, ZETA, K, B, LDB, WORK, IFAIL)}
\end{align*}
\]

In both cases WORK must be a k element array that is used as workspace. If B is a one-dimensional array (single column) then the parameter LDB can be replaced by N. See F01QDF for further details.

The first k columns of the orthogonal matrix Q can either be obtained by setting B to the first k columns of the unit matrix and using the first of the above two calls, or by calling F01QEF, which overwrites the k columns of Q on the first k columns of the array A. Q is obtained by the call:

\[
\begin{align*}
\text{CALL F01QEF('Separate', M, N, K, A, LDA, ZETA, WORK, IFAIL)}
\end{align*}
\]

As above WORK must be a k element array. If k is larger than N, then A must have been declared to have at least k columns.

Operations involving the matrix R can readily be performed by the Level 2 BLAS routines DTRSV and DTRMV (see Chapter F06), but note
that no test for near singularity of $R$ is incorporated in DTRSV. If $R$ is singular, or nearly singular then F02WUF(*) can be used to determine the singular value decomposition of $R$.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

\[
\begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F01 -- Matrix Factorizations

F01QDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01QDF performs one of the transformations

\[
T
\]

\[
B := QB \text{ or } B := Q^T B,
\]

where $B$ is an $m$ by $n_{colb}$ real matrix and $Q$ is an $m$ by $m$ orthogonal matrix, given as the product of Householder transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. Specification

```
SUBROUTINE F01QDF (TRANS, WHEREQ, M, N, A, LDA, ZETA, 1  NCOLB, B, LDB, WORK, IFAIL)
INTEGER M, N, LDA, NCOLB, LDB, IFAIL
DOUBLE PRECISION A(LDA,*), ZETA(*), B(LDB,*), WORK(*)
CHARACTER*1 TRANS, WHEREQ
```

3. Description
Q is assumed to be given by

\[ T \]
\[ Q = (Q \ Q \ \ldots \ Q) , \]
\[ n \ n-1 \ 1 \]

Q being given in the form

\[ k (I 0) \]
\[ Q = (0 T) \]
\[ k (k) \]

where

\[ T \]
\[ T = I - u u , \]
\[ k k k \]

\[ ((zeta)) \]
\[ (k) \]
\[ u = (z ) , \]
\[ k (k) \]

(zeta) is a scalar and z is an (m-k) element vector. z must be

\[ k k \]

supplied in the kth column of A in elements A(k+1,k), \ldots, A(m,k)
and (zeta) must be supplied either in A(k,k) or in ZETA(k),

\[ k \]

depending upon the parameter WHEREIT.

To obtain Q explicitly B may be set to I and pre-multiplied by Q.

This is more efficient than obtaining Q.

References


Oxford University Press.

Parameters

1: TRANS -- CHARACTER*1

On entry: the operation to be performed as follows:
TRANS = 'N' (No transpose)
Perform the operation $B := QB$.

TRANS = 'T' or 'C' (Transpose)

$T$

Perform the operation $B := Q^T B$.
Constraint: TRANS must be one of 'N', 'T' or 'C'.

2: WHERET -- CHARACTER*1
   Input

On entry: indicates where the elements of (zeta) are to be found as follows:

WHERET = 'I' (In A)
   The elements of (zeta) are in A.

WHERET = 'S' (Separate)
   The elements of (zeta) are separate from A, in ZETA.
Constraint: WHERET must be one of 'I' or 'S'.

3: M -- INTEGER
   Input

On entry: $m$, the number of rows of A. Constraint: $M \geq N$.

4: N -- INTEGER
   Input

On entry: $n$, the number of columns of A.

When $N = 0$ then an immediate return is effected.
Constraint: $N \geq 0$.

5: A(LDA,*) -- DOUBLE PRECISION array
   Input

Note: the second dimension of the array A must be at least max(1,N).
On entry: the leading $m$ by $n$ strictly lower triangular part of the array A must contain details of the matrix Q. In addition, when WHERET = 'I', then the diagonal elements of A must contain the elements of (zeta) as described under the argument ZETA below.

When WHERET = 'S', the diagonal elements of the array A are referenced, since they are used temporarily to store the (zeta), but they contain their original values on return.

6: LDA -- INTEGER
   Input

On entry:
the first dimension of the array A as declared in the (sub)program from which F01QDF is called.
Constraint: LDA \geq max(1,M).

7: ZETA(*) -- DOUBLE PRECISION array
   Input

Note: when WHERET = 'S', the dimension of the array ZETA must be greater than or equal to max(1,N). On entry: if WHERET = 'S', the array ZETA must contain the elements of
(zeta). If \( \text{ZETA}(k) = 0.0 \) then \( T \) is assumed to be \( I \) otherwise \( \text{ZETA}(k) \) is assumed to contain \( (zeta) \).

When \( \text{WHERE} = 'I' \), \( \text{ZETA} \) is not referenced.

8: \text{NCOLB} -- INTEGER
   Input
   On entry: \( \text{ncolb} \), number of columns of \( B \).
   When \( \text{NCOLB} = 0 \) then an immediate return is effected.
   Constraint: \( \text{NCOLB} \geq 0 \).

9: \text{B}(LDB,*) -- DOUBLE PRECISION array
   Input/Output
   Note: the second dimension of the array \( B \) must be at least \( \max(1,\text{NCOLB}) \).
   On entry: the leading \( m \) by \( \text{ncolb} \) part of the array \( B \) must contain the matrix to be transformed. On exit: \( B \) is overwritten by the transformed matrix.

10: \text{LDB} -- INTEGER
    Input
    On entry:
    the first dimension of the array \( B \) as declared in the (sub)program from which \text{F01QDF} is called.
    Constraint: \( \text{LDB} \geq \max(1,\text{M}) \).

11: \text{WORK}(*) -- DOUBLE PRECISION array
    Workspace
    Note: the dimension of the array \( \text{WORK} \) must be at least \( \max(1,\text{NCOLB}) \).

12: \text{IFAIL} -- INTEGER
    Input/Output
    On entry: \( \text{IFAIL} \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
    On exit: \( \text{IFAIL} = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \( \text{IFAIL} = 0 \) or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

\( \text{IFAIL} = -1 \)
   On entry \( \text{TRANS} /= 'N', 'T' \) or 'C',
   or \( \text{WHERE} /= 'I' \) or 'S',
   or \( \text{NCOLB} < 0 \),
   or \( \text{TRANS} /= 'N', 'T' \) or 'C',
   or \( \text{WHERE} /= 'I' \) or 'S',
   or \( \text{M} < 0 \),
   or \( \text{NCOLB} < 0 \),
   or \( \text{LDB} < \max(1,\text{M}) \),
   or \( \text{NCOLB} < \min(1,\text{M}) \),
   or \( \text{WORK}(*) \) not large enough.

"
or \( M < N \),

or \( N < 0 \),

or \( \text{LDA} < M \),

or \( \text{NCOLB} < 0 \),

or \( \text{LDB} < M \).

7. Accuracy

Letting \( C \) denote the computed matrix \( QB \), \( C \) satisfies the relation

\[
QC = B + E,
\]

where

\[
\|E\| \leq c(\text{epsilon})\|B\|,
\]

and (\text{epsilon}) the machine precision (see \text{X02AJF}(*)), \( c \) is a modest function of \( m \) and \( \|\cdot\| \) denotes the spectral (two) norm. An equivalent result holds for the computed matrix \( QB \). See also Section 7 of \text{F01QCF}.

8. Further Comments

The approximate number of floating-point operations is given by

\[
2n(2m-n)\text{colb}.
\]

9. Example

To obtain the matrix \( QB \) for the matrix \( B \) given by

\[
\begin{pmatrix}
1.1 & 0.00 \\
0.9 & 0.00 \\
0.6 & 1.32 \\
0.0 & 1.10 \\
-0.8 & -0.26
\end{pmatrix}
\]

following the QR factorization of the 5 by 3 matrix \( A \) given by

\[
\begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**F01QEF**

**F01QEF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

F01QEF returns the first ncolq columns of the real m by m orthogonal matrix Q, where Q is given as the product of Householder transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. **Specification**

```
SUBROUTINE F01QEF (WHERE, M, N, NCOLQ, A, LDA, ZETA, WORK, IFAIL)
   INTEGER M, N, NCOLQ, LDA, IFAIL
   DOUBLE PRECISION A(LDA,*), ZETA(*), WORK(*)
   CHARACTER*1 WHERE

3. **Description**

Q is assumed to be given by

\[
T
Q = (Q \quad \ldots \quad Q ),
\]

\[n \quad n-1 \quad 1\]

Q being given in the form

\[k \]

\[
(I \ 0 )
Q = (0 \ T )
\]

\[k \ ( \ k)\]

where

\[
T
T = I - u u ,
\]
\[(\zeta)\kern0pt\text{ is a scalar and } z \text{ is an } (m-k) \text{ element vector. } z \text{ must be supplied in the } k\text{th column of } A \text{ in elements } A(k+1,k),\ldots,A(m,k) \text{ and } (\zeta) \text{ must be supplied either in } A(k,k) \text{ or in } ZETA(k), \text{ depending upon the parameter WHERE.}
\]

4. References


5. Parameters

1: WHERE -- CHARACTER*1 Input
   On entry: indicates where the elements of (\zeta) are to be found as follows:
   WHERE = 'I' (In A)
   The elements of (\zeta) are in A.
   WHERE = 'S' (Separate)
   The elements of (\zeta) are separate from A, in ZETA. Constraint: WHERE must be one of 'I' or 'S'.

2: M -- INTEGER Input
   On entry: m, the number of rows of A. Constraint: M \geq N.

3: N -- INTEGER Input
   On entry: n, the number of columns of A. Constraint: N \geq 0.

4: NCOLQ -- INTEGER Input
   On entry: ncolq, the required number of columns of Q. Constraint: 0 \leq NCOLQ \leq M.
   When NCOLQ = 0 then an immediate return is effected.

5: A(LDA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least
max(1,N,NCOLQ).
On entry: the leading m by n strictly lower triangular part of the array A must contain details of the matrix Q. In addition, when WHERET = 'I', then the diagonal elements of A must contain the elements of (zeta) as described under the argument ZETA below. On exit: the first NCOLQ columns of the array A are overwritten by the first NCOLQ columns of the m by m orthogonal matrix Q. When N = 0 then the first NCOLQ columns of A are overwritten by the first NCOLQ columns of the identity matrix.

6: LDA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F01QEF is called.
Constraint: LDA >= max(1,M).

7: ZETA(*) -- DOUBLE PRECISION array Input
Note: the dimension of the array ZETA must be at least max(1,N).
On entry: with WHERET = 'S', the array ZETA must contain the elements of (zeta). If ZETA(k) = 0.0 then T is assumed to be I, otherwise ZETA(k) is assumed to contain (zeta).

When WHERET = 'I', the array ZETA is not referenced.

8: WORK(*) -- DOUBLE PRECISION array Workspace
Note: the dimension of the array WORK must be at least max(1,NCOLQ).

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1
On entry WHERET /= 'I' or 'S',
or \( M < N \),

or \( N < 0 \),

or \( \text{NCOLQ} < 0 \) or \( \text{NCOLQ} > M \),

or \( \text{LDA} < M \).

7. Accuracy

The computed matrix \( Q \) satisfies the relation

\[
Q = P + E,
\]

where \( P \) is an exactly orthogonal matrix and

\[
||E|| \leq c(\epsilon),
\]

(\( \epsilon \)) is the machine precision (see X02AJF(*)), \( c \) is a modest function of \( m \) and \( |||\cdot||| \) denotes the spectral (two) norm. See also Section 7 of F01QCF.

8. Further Comments

The approximate number of floating-point operations required is given by

\[
2^{-n][(3m-n)(2\text{NCOLQ}-n)n(\text{NCOLQ}-n)}, \quad \text{NCOLQ}>n,
\]

\[
3^{2}2^{-n}(3m-\text{NCOLQ}), \quad \text{NCOLQ}\leq n.
\]

9. Example

To obtain the 5 by 5 orthogonal matrix \( Q \) following the QR factorization of the 5 by 3 matrix \( A \) given by

\[
A = \begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F01 -- Matrix Factorizations

F01RCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01RCF finds the QR factorization of the complex m by n matrix A, where m\geq n.

2. Specification

```fortran
SUBROUTINE F01RCF (M, N, A, LDA, THETA, IFAIL)
INTEGER M, N, LDA, IFAIL
COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*)
```

3. Description

The m by n matrix A is factorized as

\[
A = \begin{cases} 
Q(0) & \text{when } m > n, \\
QR & \text{when } m = n,
\end{cases}
\]

where Q is an m by m unitary matrix and R is an n by n upper triangular matrix with real diagonal elements.

The factorization is obtained by Householder’s method. The kth transformation matrix, Q, which is used to introduce zeros into the kth column of A is given in the form

\[
Q^k = \begin{pmatrix} I & 0 \\
0 & T \end{pmatrix},
\]

where

\[
T = I - \alpha \frac{u_k u_k^T}{\mathbf{u}_k^T \mathbf{u}_k},
\]

\[
\alpha = \frac{1}{\mathbf{u}_k^T \mathbf{u}_k}.
\]
u = (z_k),
k ( k )

(gamma) is a scalar for which Re (gamma) = 1.0, (zeta) is a real scalar and z is an (m-k) element vector. (gamma), (zeta) and z are chosen to annihilate the elements below the triangular part of A and to make the diagonal elements real.

The scalar (gamma) and the vector u are returned in the kth element of the array THETA and in the kth column of A, such that (theta), given by

( theta ) = ( (zeta), Im(gamma) ),
k k k k

is in THETA(k) and the elements of z are in a_k, ..., a_m. The elements of R are returned in the upper triangular part of A.

Q is given by

Q = ( Q_n Q_{n-1} ... Q_1 ).
n n-1 1

A good background description to the QR factorization is given in Dongarra et al [1], but note that this routine is not based upon LINPACK routine ZQRDC.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER Input
   On entry: n, the number of columns of A. Constraint: N >= 0.
When N = 0 then an immediate return is effected.

3: A(LDA,*) -- COMPLEX(KIND(1.0D0)) array Input/Output
   Note: the second dimension of the array A must be at least \( \max(1,N) \).
   On entry: the leading m by n part of the array A must
   contain the matrix to be factorized. On exit: the n by n
   upper triangular part of A will contain the upper triangular
   matrix R, with the imaginary parts of the diagonal elements
   set to zero, and the m by n strictly lower triangular part
   of A will contain details of the factorization as described
   above.

4: LDA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the
   (sub)program from which F01RCF is called.
   Constraint: LDA >= \( \max(1,M) \).

5: THETA(*) -- COMPLEX(KIND(1.0D0)) array Output
   Note: the dimension of the array THETA must be at least
   \( \max(1,N) \).
   On exit: the scalar (theta) for the kth transformation. If
   \( k \) T =I then THETA(k) = 0.0; if
   \( k \)
   \[ ((\alpha) 0) \]
   \[ \begin{array}{c}
   0 \\
   \alpha
   \end{array} \] Re(\alpha)<0.0,
   \( k \)
   then THETA(k)=(\alpha); otherwise THETA(k) contains THETA(k)
   as described in Section 3 and Re(\theta(k)) is always in the
   range (1.0, \( \sqrt{2.0} \)).

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).
IFAIL=-1
   On entry M < N,
   or N < 0,
   or LDA < M.

7. Accuracy
The computed factors Q and R satisfy the relation

   (R)
   Q(0)=A+E,

where

   ||E||<=c(epsilon)||A||,

(epsilon) being the machine precision, c is a modest function of m and n and ||.|| denotes the spectral (two) norm.

8. Further Comments
The approximate number of real floating-point operations is given by

   2
   8n (3m-n)/3.

Following the use of this routine the operations

   H
   B:=QB and B:=Q B,

where B is an m by k matrix, can be performed by calls to F01RDF. The operation B:=QB can be obtained by the call:

   IFAIL = 0
   CALL F01RDF('No conjugate', 'Separate', M, N, A, LDA, THETA, *      K, B, LDB, WORK, IFAIL)

   H

and B:=Q B can be obtained by the call:

   IFAIL = 0
   CALL F01RDF('Conjugate', 'Separate', M, N, A, LDA, THETA, *      K, B, LDB, WORK, IFAIL)

In both cases WORK must be a k element array that is used as workspace. If B is a one-dimensional array (single column) then
the parameter LDB can be replaced by M. See F01RDF for further details.

The first k columns of the unitary matrix Q can either be obtained by setting B to the first k columns of the unit matrix and using the first of the above two calls, or by calling F01REF, which overwrites the k columns of Q on the first k columns of the array A. Q is obtained by the call:

    CALL F01REF('Separate', M, N, K, A, LDA, THETA, WORK, IFAIL)

As above, WORK must be a k element array. If k is larger than n, then A must have been declared to have at least k columns.

Operations involving the matrix R can readily be performed by the Level 2 BLAS routines ZTRSV and ZTRMV (see Chapter F06), but note that no test for near singularity of R is incorporated in ZTRSV. If R is singular, or nearly singular, then F02XUF(*) can be used to determine the singular value decomposition of R.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

\[
\begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.0i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.4 & -0.4+0.4i & 1.8 \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
where $B$ is an $m$ by $ncolb$ complex matrix and $Q$ is an $m$ by $m$ unitary matrix, given as the product of Householder transformation matrices.

This routine is intended for use following F01RCF or F01RFF(*).

2. Specification

```fortran
SUBROUTINE F01RDF (TRANS, WHERET, M, N, A, LDA, THETA, NCOLB, B, LDB, WORK, IFAIL)
INTEGER M, N, LDA, NCOLB, LDB, IFAIL
COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), B(LDB,*), WORK(*)
CHARACTER*1 TRANS, WHERET
```

3. Description

The unitary matrix $Q$ is assumed to be given by

$$
Q = (Q Q \ldots Q),
$$

$\begin{bmatrix}
  Q & \ldots & Q \\
  n & n-1 & 1
\end{bmatrix}$

$Q$ being given in the form

$$
\begin{bmatrix}
  I & 0 \\
  0 & T
\end{bmatrix},
$$

$k$

where

$$
T = I - (\gamma) u u^T,
$$

$k$

$$
((\zeta))
$$

$k$

$u = (z \ldots ),

$k$

$(\gamma)$ is a scalar for which $Re(\gamma) = 1.0$, $(\zeta)$ is a real scalar and $z$ is an $(m-k)$ element vector.

$z$ must be supplied in the $k$th column of $A$ in elements

$k$

$a, \ldots, a$ and $(\theta)$, given by

$$
B := QB $ or $ B := QB,
$$

where

$B$ is an $m$ by $ncolb$ complex matrix and $Q$ is an $m$ by $m$ unitary matrix, given as the product of Householder transformation matrices.
must be supplied either in $a$ or in $\text{THETA}(k)$, depending upon $k$.

To obtain $Q$ explicitly $B$ may be set to $I$ and pre-multiplied by $Q$. This is more efficient than obtaining $Q$. Alternatively, F01REF may be used to obtain $Q$ overwritten on $A$.

4. References


5. Parameters

1: TRANS -- CHARACTER*1 Input
   On entry: the operation to be performed as follows:
   TRANS = 'N' (No transpose)
   Perform the operation $B := QB$.
   TRANS = 'C' (Conjugate transpose)
   Perform the operation $B := QB$.
   Constraint: TRANS must be one of 'N' or 'C'.

2: WHERET -- CHARACTER*1 Input
   On entry: the elements of $(\theta)$ are to be found as follows:
   WHERET = 'I' (In A)
   The elements of $(\theta)$ are in $A$.
   WHERET = 'S' (Separate)
   The elements of $(\theta)$ are separate from $A$, in $\text{THETA}$. Constraint: WHERET must be one of 'I' or 'S'.

3: M -- INTEGER Input
   On entry: $m$, the number of rows of $A$. Constraint: $M >= N$.

4: N -- INTEGER Input
   On entry: $n$, the number of columns of $A$. Constraint: $N >= 0$.

   When $N = 0$ then an immediate return is effected.
5: A(LDA,*) -- COMPLEX(KIND(1.0D)) array  
   Input  
   Note: the second dimension of the array A must be at least 
       max(1,N).  
   On entry: the leading m by n strictly lower triangular part 
       of the array A must contain details of the matrix Q. In 
       addition, when WHERET = 'I', then the diagonal elements of A 
       must contain the elements of (theta) as described under the 
       argument THETA below. 

When WHERET = 'S', then the diagonal elements of the array A 
are referenced, since they are used temporarily to store the 
(zeta), but they contain their original values on return. 

6: LDA -- INTEGER Input  
   On entry: 
       the first dimension of the array A as declared in the 
       (sub)program from which F01RDF is called. 
   Constraint: LDA >= max(1,M). 

7: THETA(*) -- COMPLEX(KIND(1.0D)) array  
   Input  
   Note: the dimension of the array THETA must be at least 
       max(1,N).  
   On entry: with WHERET = 'S', the array THETA must contain 
       the elements of (theta). If THETA(k)=0.0 then T is assumed 
       k 
       to be I; if THETA(k)=(alpha), with Re(alpha)<0.0, then T is 
       k 
       assumed to be of the form 

       \[ \begin{pmatrix} \alpha & 0 \\ 0 & 1 \end{pmatrix} \] 

       k

   otherwise THETA(k) is assumed to contain (theta) given by 

   \[ \begin{pmatrix} \zeta & \text{Im}(\gamma) \\ \text{Re}(\gamma) & \text{Re}(\gamma) \end{pmatrix} \] 

   k  k  k

   When WHERET = 'I', the array THETA is not referenced, and 
   may be dimensioned of length 1. 

8: NCOLB -- INTEGER Input  
   On entry: ncolb, the number of columns of B. Constraint: 
       NCOLB >= 0. 
   When NCOLB = 0 then an immediate return is effected. 

9: B(LDB,*) -- COMPLEX(KIND(1.0D)) array  
   Input/Output  
   Note: the second dimension of the array B must be at least 
       max(1,NCOLB). 
   On entry: the leading m by ncolb part of the array B must 
       contain the matrix to be transformed. On exit: B is
overwritten by the transformed matrix.

10: LDB -- INTEGER Input
   On entry:
   the first dimension of the array B as declared in the
   (sub)program from which F01RDF is called.
   Constraint: LDB >= max(1,M).

11: WORK(*) -- COMPLEX(KIND(1.0D)) array Workspace
   Note: the dimension of the array WORK must be at least
   max(1,NCOLB).

12: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
   On entry TRANS /= 'N' or 'C',
   or WHERET /= 'I' or 'S',
   or M < N,
   or N < 0,
   or LDA < M,
   or NCOLB < 0,
   or LDB < M.

7. Accuracy

Letting C denote the computed matrix Q B, C satisfies the relation

QC=B+E,

where
\[ ||E|| \leq c(\text{epsilon}) ||B||, \]

(\text{epsilon}) being the machine precision, \(c\) is a modest function of \(m\) and \(||.||\) denotes the spectral (two) norm. An equivalent result holds for the computed matrix \(QB\). See also Section 7 of F01RCF.

8. Further Comments

The approximate number of real floating-point operations is given by \(8n(2m-n)\text{ncolb}\).

9. Example

\[
H
\]

To obtain the matrix \(QB\) for the matrix \(B\) given by

\[
\begin{pmatrix}
-0.55+1.05i & 0.45+1.05i \\
0.49+0.93i & 1.09+0.13i \\
0.56-0.16i & 0.64+0.16i \\
0.39+0.23i & -0.39-0.23i \\
1.13+0.83i & -1.13+0.77i
\end{pmatrix}
\]

following the QR factorization of the 5 by 3 matrix \(A\) given by

\[
\begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.0i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.4 & -0.4+0.4i & 1.8 \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations

F01REF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01REF returns the first \(\text{ncolq}\) columns of the complex \(m\) by \(m\) unitary matrix \(Q\), where \(Q\) is given as the product of Householder
transformation matrices.

This routine is intended for use following F01RCF.

2. Specification

```fortran
SUBROUTINE F01REF (WHERET, M, N, NCOLQ, A, LDA, THETA,
                   WORK, IFAIL)
  INTEGER M, N, NCOLQ, LDA, IFAIL
  COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), WORK(*)
  CHARACTER*1 WHERET
```

3. Description

The unitary matrix Q is assumed to be given by

\[
Q_H = (Q_{n-1} Q_{n-2} \ldots Q_1),
\]

Q being given in the form

\[
Q_k = \begin{pmatrix}
I & 0 \\
0 & T_k
\end{pmatrix},
\]

where

\[
T_k = I - (\gamma_k) u_k u_k^* - (\zeta_k) u_k^* u_k,
\]

where

\[
\gamma_k, \zeta_k, u_k
\]

\(\gamma_k\) is a scalar for which \(\Re(\gamma_k) = 1.0\), \(\zeta_k\) is a real scalar and \(z_k\) is an \((m-k)\) element vector.

\(z_k\) must be supplied in the \(k\)th column of \(A\) in elements \(a_k, \ldots, a_{m-k}\) and \((\theta_k)\), given by

\[
(\theta_k) = (\zeta_k, \Im(\gamma_k)),
\]
must be supplied either in $a_{k,k}$ or in $\text{THETA}(k)$ depending upon the $k,k$ parameter $\text{WERET}$.

4. References


5. Parameters

1: $\text{WERET} \quad \text{CHARACTER*1} \quad \text{Input}$
   On entry: the elements of (theta) are to be found as follows:
   $\text{WERET} = 'I'$ (In $A$)
   The elements of (theta) are in $A$.
   $\text{WERET} = 'S'$ (Separate)
   The elements of (theta) are separate from $A$, in $\text{THETA}$.
   Constraint: $\text{WERET}$ must be one of 'I' or 'S'.

2: $M \quad \text{INTEGER} \quad \text{Input}$
   On entry: $m$, the number of rows of $A$. Constraint: $M \geq N$.

3: $N \quad \text{INTEGER} \quad \text{Input}$
   On entry: $n$, the number of columns of $A$. Constraint: $N \geq 0$.

4: $\text{NCOLOQ} \quad \text{INTEGER} \quad \text{Input}$
   On entry: $\text{ncolq}$, the required number of columns of $Q$.
   Constraint: $0 \leq \text{NCOLOQ} \leq M$.
   When $\text{NCOLOQ} = 0$ then an immediate return is effected.

5: $A(LDA,*) \quad \text{COMPLEX(KIND(1.0D)) array} \quad \text{Input/Output}$
   Note: the second dimension of the array $A$ must be at least
   $\max(1,N,\text{NCOLOQ})$.
   On entry: the leading $m$ by $n$ strictly lower triangular part
   of the array $A$ must contain details of the matrix $Q$. In
   addition, when $\text{WERET} = 'I'$, then the diagonal elements of $A$
   must contain the elements of (theta) as described under the
   argument $\text{THETA}$ below. On exit: the first $\text{NCOLOQ}$ columns of
   the array $A$ are overwritten by the first $\text{NCOLOQ}$ columns of
   the $m$ by $m$ unitary matrix $Q$. When $N = 0$ then the first $\text{NCOLOQ}$
   columns of $A$ are overwritten by the first $\text{NCOLOQ}$ columns of
   the unit matrix.

6: $LDA \quad \text{INTEGER} \quad \text{Input}$
On entry:
the first dimension of the array A as declared in the
(sub)program from which F01REF is called.
Constraint: LDA >= max(1,M).

7: THETA(*) -- COMPLEX(KIND(1.0D)) array Input
Note: the dimension of the array THETA must be at least
max(1,N).
On entry: if WHERET = 'S', the array THETA must contain the
elements of (theta). If THETA(k)=0.0 then T is assumed to
be I; if THETA(k)=(alpha), with Re(alpha)<0.0, then T is
assumed to be of the form
\[
\begin{pmatrix}
\alpha & 0 \\
0 & 1
\end{pmatrix}
\]
k
otherwise THETA(k) is assumed to contain (theta) given by
\[
(\theta_k) = (\zeta_k, \text{Im}(\gamma_k)).
\]
k

When WHERET = 'I', the array THETA is not referenced.

8: WORK(*) -- COMPLEX(KIND(1.0D)) array Workspace
Note: the dimension of the array WORK must be at least
max(1,NCOLQ).

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
On entry WHERET /= 'I' or 'S',
or M < N,
or N < 0,
or \( \text{NCOLQ} < 0 \) or \( \text{NCOLQ} > M \),

or \( \text{LDA} < M \).

7. Accuracy

The computed matrix \( Q \) satisfies the relation

\[
Q = P + E,
\]

where \( P \) is an exactly unitary matrix and

\[
||E|| = c(\text{epsilon}),
\]

(\( \text{epsilon} \)) being the machine precision, \( c \) is a modest function of \( m \) and \( |||\cdot||| \) denotes the spectral (two) norm. See also Section 7 of F01RCF.

8. Further Comments

The approximate number of real floating-point operations required is given by

\[
8 - n((3m-n)(2\text{ncolq}-n)-n(\text{ncolq}-n)), \quad \text{ncolq}>n
\]

3

\[
8 2 - n(3m-\text{ncolq}), \quad \text{ncolq} \leq n
\]

3

9. Example

To obtain the 5 by 5 unitary matrix \( Q \) following the QR factorization of the 5 by 3 matrix \( A \) given by

\[
\begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.4i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.4 & -0.4+0.4i & 1.8 \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagMatrixOperationsPackage (NAGF01)

Exports:

\begin{verbatim}
<table>
<thead>
<tr>
<th>Exports</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>f01brf</td>
<td>f01bsf f01maf f01mcf f01qcf f01qdf f01qef f01rcf f01rdf f01ref</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textit{package NAGF01 NagMatrixOperationsPackage}

\textbf{Exports} == \textbf{Implementation} where

\texttt{S} => \texttt{Symbol}

\texttt{FOP} => \texttt{FortranOutputStackPackage}

\textbf{Exports} => with

\texttt{f01brf} : \texttt{(Integer,Integer,Integer,Integer,DoubleFloat,Boolean,Boolean,List Boolean,Matrix DoubleFloat,Matrix Integer,Integer) -> Result}

\texttt{f01brf(n,nz,licn,lirn,pivot,lblock,grow,abort,a,irn,icn,ifail)}

\texttt{++ factorizes a real sparse matrix. The routine either forms}

\texttt{++ the LU factorization of a permutation of the entire matrix, or,}

\texttt{++ optionally, first permutes the matrix to block lower triangular}

\texttt{++ form and then only factorizes the diagonal blocks.}

\texttt{++ See \textit{Manual Page}{manpageXXf01brf}.}

\texttt{f01bsf} : \texttt{(Integer,Integer,Integer,Matrix Integer,Matrix Integer,Integer,Matrix Integer,Matrix Integer,DoubleFloat,Integer) -> Result}

\texttt{f01bsf(n,nz,licn,jvect,icn,ikeep,grow,eta,abort,avals,ifail)}

\texttt{++ factorizes a real sparse matrix using the pivotal sequence}

\texttt{++ previously obtained by F01BRF when a matrix of the same sparsity}
++ pattern was factorized.
++ See \downlink{Manual Page}{manpageXXf01bsf}.
f01maf : (Integer,Integer,Integer,Integer,Integer,_
    List Boolean,Matrix DoubleFloat,Matrix Integer,Matrix Integer,_
    DoubleFloat,DoubleFloat,Integer) -> Result
++ f01maf(n,nz,licn,lirn,abort,avals,irn,icn,droptl,densw,ifail)
++ computes an incomplete Cholesky factorization of a real
++ sparse symmetric positive-definite matrix A.
++ See \downlink{Manual Page}{manpageXXf01maf}.
f01mcf : (Integer,Matrix DoubleFloat,Integer,Matrix Integer,_
    Integer) -> Result
++ f01mcf(n,avals,lal,nrow,ifail)
++ computes the Cholesky factorization of a real symmetric
++ positive-definite variable-bandwidth matrix.
++ See \downlink{Manual Page}{manpageXXf01mcf}.
f01qcf : (Integer,Integer,Integer,Matrix DoubleFloat,_
    Integer) -> Result
++ f01qcf(m,n,lda,a,ifail)
++ finds the QR factorization of the real m by n matrix A,
++ where m>=n.
++ See \downlink{Manual Page}{manpageXXf01qcf}.
f01qdf : (String,String,Integer,Integer,_
    Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,Integer,_
    Matrix DoubleFloat,Integer) -> Result
++ f01qdf(trans,wheret,m,n,a,lda,zeta,ncolb,ldb,b,ifail)
++ performs one of the transformations
++ See \downlink{Manual Page}{manpageXXf01qdf}.
f01qef : (String,Integer,Integer,Integer,_
    Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f01qef(wheret,m,n,ncolq,lda,zeta,a,ifail)
++ returns the first ncolq columns of the real m by m
++ orthogonal matrix Q, where Q is given as the product of
++ Householder transformation matrices.
++ See \downlink{Manual Page}{manpageXXf01qef}.
f01rcf : (Integer,Integer,Integer,Matrix Complex DoubleFloat,_
    Integer) -> Result
++ f01rcf(m,n,lda,a,ifail)
++ finds the QR factorization of the complex m by n matrix A,
++ where m>=n.
++ See \downlink{Manual Page}{manpageXXf01rcf}.
f01rdf : (String,String,Integer,Integer,_
    Matrix Complex DoubleFloat,Integer,Matrix Complex DoubleFloat,Integer,Integer,_
    Matrix Complex DoubleFloat,Integer) -> Result
++ f01rdf(trans,wheret,m,n,a,lda,theta,ncolb,ldb,b,ifail)
++ performs one of the transformations
++ See \downlink{Manual Page}{manpageXXf01rdf}.
f01ref : (String,Integer,Integer,Integer,_
    Integer,Matrix Complex DoubleFloat,Matrix Complex DoubleFloat,Integer) -> Result
++ f01ref(wheret,m,n,ncolq,lda,theta,a,ifail)
**returns the first ncolq columns of the complex m by m unitary matrix Q, where Q is given as the product of Householder transformation matrices.**

See \downlink{Manual Page}{manpageXXf01ref}.

Implementation => add

```lisp
import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(String)
import AnyFunctions1(List Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(Matrix Integer)

f01brf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
lrnArg:Integer,pivotArg:DoubleFloat,lblockArg:Boolean,_
growArg:Boolean,abortArg:List Boolean,aArg:Matrix DoubleFloat,_
irnArg:Matrix Integer,icnArg:Matrix Integer,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f01brf",_
"w"::S,"idisp"::S,"a"::S_,
"lrn"::S,"icn"::S,"iw"::S]$Lisp_,
[["double"::S,"pivot"::S,["w"::S,"n"::S]$Lisp_,
["a"::S,"licn"::S]$Lisp]$Lisp_,
["ikeep"::S,["*"::S,$Lisp,"n"::S]$Lisp]$Lisp_,
["idisp"::S,10$Lisp]$Lisp,$Lisp,["lrn"::S,"lrn"::S]$Lisp_,
["icn"::S,"licn"::S]$Lisp_,
]$Lisp_,
"icn"::S,"ifail"::S]$Lisp_,
[[nArg::Any,nzArg::Any,licnArg::Any,lrnArg::Any,pivotArg::Any_,
lblockArg::Any,growArg::Any,ifailArg::Any,abortArg::Any_,
aArg::Any,lrnArg::Any,icnArg::Any ]])
```
fake code
CHAPTER 15. CHAPTER N

[$Lisp,$
"icn"::S,"droptl"::S,"densw"::S,"ifail"::S]$Lisp,
[(nArg::Any,nzArg::Any,licnArg::Any,lirnArg::Any,droptlArg::Any,
denswArg::Any,ifailArg::Any,abortArg::Any,avalsArg::Any,
irnArg::Any,icnArg::Any )]
@List Any]$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f01mcf(nArg:Integer,avalsArg:Matrix DoubleFloat,lalArg:Integer,
nrowArg:Matrix Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,
"f01mcf",
["al":S,"d":S]$Lisp,
["double":S,["avals":S,"lal":S]$Lisp,["al":S,"lal":S]$Lisp,
,"n":S,$Lisp,$Lisp,
,"ifail":S]$Lisp,
]$Lisp,
["al":S,"d":S,"ifail":S]$Lisp,
[(nArg::Any,lalArg::Any,ifailArg::Any,avalsArg::Any,nrowArg::Any )]
@List Any]$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f01qcf(mArg:Integer,nArg:Integer,ldaArg:Integer,
aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,
"f01qcf",
["m":S,"m":S,"n":S,"lda":S,"ifail":S,"n":S,"a":S]$Lisp,
["n":S],$Lisp,
["double":S,["n":S]$Lisp,["a":S,"lda":S,"n":S]$Lisp,
]$Lisp,
]$Lisp,
]$Lisp,
["zeta":S,"a":S,"ifail":S]$Lisp,
[(mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any )]
@List Any]$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f01qdf(transArg:String,wheretArg:String,mArg:Integer,
aArg:Matrix DoubleFloat,ldaArg:Integer,
zetaArg:Matrix DoubleFloat,ncolbArg:Integer,ldbArg:Integer,
bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,
"f01qdf",
"ifail":S,"a":S,"zeta":S,"b":S,"work":S]$Lisp,
["work":S]$Lisp,
]$Lisp,
[(invokeNagman(NIL$Lisp, 
"f01qef", 
["work"::S]$Lisp, 
["character"::S,"wheret"::S]$Lisp ]$Lisp, 
["a"::S,"ifail"::S]$Lisp, 
[(transArg::Any,wheretArg::Any,mArg::Any,nArg::Any,ldaArg::Any, 
ncolqArg::Any,ldbArg::Any,ifailArg::Any,aArg::Any, 
zetaArg::Any,bArg::Any )] 
@List Any$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

f01rcf(mArg: Integer, nArg: Integer, ldaArg: Integer, 
aArg: Matrix Complex DoubleFloat, ifailArg: Integer): Result == 
[(invokeNagman(NIL$Lisp, 
"f01rcf", 
["theta"::S]$Lisp, 
["integer"::S,"m"::S,"n"::S,"lda"::S,"ifail"::S]$Lisp, 
["double complex"::S,["theta"::S,"n"::S]$Lisp, 
["a"::S,"lda"::S,"n"::S]$Lisp]$Lisp]$Lisp, 
["theta"::S,"a"::S,"ifail"::S]$Lisp, 
[(mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any ]), 
@List Any$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

f01rdf(transArg: String, wheretArg: String, mArg: Integer, 
nArg: Integer, aArg: Matrix Complex DoubleFloat, ldaArg: Integer,)
\begin{verbatim}
thetaArg: Matrix Complex DoubleFloat, ncolbArg: Integer, ldbArg: Integer, 
  bArg: Matrix Complex DoubleFloat, ifailArg: Integer): Result ==
  [[invokeNagman(NIL$Lisp, 
    "f01rdf", 
    ["trans": S, "wheret": S, "m": S, "n": S, "lda": S, 
      "ncolb": S, "ldb": S, "ifail": S, "a": S, "theta": S, 
      "b": S, "work": S]@Lisp, ["work": S]@Lisp, 
    [[integer": S, "m": S, "n": S, "lda": S, 
      "ncolb": S, "ldb": S, "ifail": S]@Lisp, 
    ["character": S, "trans": S, "wheret": S]@Lisp, 
    ["double complex": S, ["a": S, "lda": S, "n": S]@Lisp, 
    ["theta": S, "n": S]@Lisp, ["b": S, "ldb": S, "ncolb": S]@Lisp, 
    ["work": S, "ncolb": S]@Lisp, ["work": S]@Lisp, 
    ["b": S, "ifail": S]@Lisp], 
  ["trans": S, "wheret": S]@Lisp, 
  ["double complex": S, ["a": S, "lda": S, "n": S]@Lisp, 
  ["theta": S, "n": S]@Lisp, ["b": S, "ldb": S, "ncolb": S]@Lisp, 
  ["work": S, "ncolb": S]@Lisp, ["work": S]@Lisp, 
  ["b": S, "ifail": S]@Lisp, 
  [[transArg:: Any, wheretArg:: Any, nArg:: Any, aArg:: Any, 
    ldaArg:: Any, ncolbArg:: Any, ldbArg:: Any, ifailArg:: Any, aArg:: Any, 
    thetaArg:: Any, bArg:: Any ]@List Any]@Lisp)@Lisp) 
pretend List (Record(key: Symbol, entry: Any))@$Result

f01ref(wheretArg:: String, mArg: Integer, nArg: Integer, 
  ncolqArg: Integer, ldaArg: Integer, thetaArg: Matrix Complex DoubleFloat, 
  aArg: Matrix Complex DoubleFloat, ifailArg: Integer): Result ==
  [[invokeNagman(NIL$Lisp, 
    "f01ref", 
    ["wheret": S, "m": S, "n": S, "ncolq": S, "lda": S, 
      "ifail": S, "theta": S, "a": S, "work": S]@Lisp, 
    ["work": S]@Lisp, 
    [[integer": S, "m": S, "n": S, "ncolq": S, "lda": S, 
      "ifail": S]@Lisp, 
    ["character": S, "wheret": S]@Lisp, 
    ["double complex": S, ["theta": S, "m": S]@Lisp, 
    ["a": S, "lda": S, "n": S]@Lisp, ["work": S, "ncolq": S]@Lisp, 
    ["work": S]@Lisp, 
    ["a": S, "ifail": S]@Lisp, 
    [[wheretArg:: Any, mArg:: Any, nArg:: Any, ncolqArg:: Any, ldaArg:: Any, 
      ifailArg:: Any, thetaArg:: Any, aArg:: Any ]@List Any]@Lisp)@Lisp) 
pretend List (Record(key: Symbol, entry: Any))@$Result
\end{verbatim}
This package uses the NAG Library to perform optimization. An optimization problem involves minimizing a function (called the objective function) of several variables, possibly subject to restrictions on the values of the variables defined by a set of constraint functions. The routines in the NAG Foundation Library are concerned with function minimization only, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by -1.
2.1.3. Minimization subject to bounds on the variables
2.1.4. Minimization subject to linear constraints
2.1.5. Minimization subject to nonlinear constraints
2.2. Geometric Representation and Terminology
2.2.1. Gradient vector
2.2.2. Hessian matrix
2.2.3. Jacobian matrix; matrix of constraint normals
2.3. Sufficient Conditions for a Solution
2.3.1. Unconstrained minimization
2.3.2. Minimization subject to bounds on the variables
2.3.3. Linearly-constrained minimization
2.3.4. Nonlinearly-constrained minimization
2.4. Background to Optimization Methods
2.4.1. Methods for unconstrained optimization
2.4.2. Methods for nonlinear least-squares problems
2.4.3. Methods for handling constraints
2.5. Scaling
2.5.1. Transformation of variables
2.5.2. Scaling the objective function
2.5.3. Scaling the constraints
2.6. Analysis of Computed Results
2.6.1. Convergence criteria
2.6.2. Checking results
2.6.3. Monitoring progress
2.6.4. Confidence intervals for least-squares solutions
1. Scope of the Chapter

An optimization problem involves minimizing a function (called the objective function) of several variables, possibly subject to restrictions on the values of the variables defined by a set of constraint functions. The routines in the NAG Foundation Library are concerned with function minimization only, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by -1.

This introduction is only a brief guide to the subject of optimization designed for the casual user. Anyone with a difficult or protracted problem to solve will find it beneficial to consult a more detailed text, such as Gill et al [5] or Fletcher [3].

Readers who are unfamiliar with the mathematics of the subject may find some sections difficult at first reading; if so, they should concentrate on Sections 2.1, 2.2, 2.5, 2.6 and 3.

2. Background to the Problems

2.1. Types of Optimization Problems

Solution of optimization problems by a single, all-purpose, method is cumbersome and inefficient. Optimization problems are therefore classified into particular categories, where each category is defined by the properties of the objective and constraint functions, as illustrated by some examples below.

<table>
<thead>
<tr>
<th>Properties of Objective Function</th>
<th>Properties of Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonlinear</td>
<td>Nonlinear</td>
</tr>
<tr>
<td>Sums of squares of</td>
<td>Sparse linear</td>
</tr>
</tbody>
</table>
nonlinear functions

Quadratic Linear

Sums of squares of linear Bounds
functions

Linear None

For instance, a specific problem category involves the minimization of a nonlinear objective function subject to bounds on the variables. In the following sections we define the particular categories of problems that can be solved by routines contained in this Chapter.

2.1.1. Unconstrained minimization

In unconstrained minimization problems there are no constraints on the variables. The problem can be stated mathematically as follows:

\[
\text{minimize } F(x) \\
x
\]

\[x \in \mathbb{R}^n, \text{ that is, } x=(x_1, x_2, \ldots, x_n).
\]

2.1.2. Nonlinear least-squares problems

Special consideration is given to the problem for which the function to be minimized can be expressed as a sum of squared functions. The least-squares problem can be stated mathematically as follows:

\[
\begin{cases}
\{ m \\
\{ T -- 2 \\
\end{cases}
\text{minimize } \{ f(f=) \ f(x) \}, \ x \in \mathbb{R}^n \\
x \{ -- i \\
\} \{ i=1 \\
\}
\]

where the ith element of the m-vector f is the function f(x). i

2.1.3. Minimization subject to bounds on the variables

These problems differ from the unconstrained problem in that at least one of the variables is subject to a simple restriction on its value, e.g. \(x \leq 10\), but no constraints of a more general form
are present.

The problem can be stated mathematically as follows:

\[
\begin{align*}
\text{minimize } & F(x), \ x \text{ is in } \mathbb{R} \\
\text{subject to } & l \leq x \leq u, \ i=1,2,\ldots,n.
\end{align*}
\]

This format assumes that upper and lower bounds exist on all the variables. By conceptually allowing \( u = \infty \) and \( l = -\infty \) all the variables need not be restricted.

2.1.4. Minimization subject to linear constraints

A general linear constraint is defined as a constraint function that is linear in more than one of the variables, e.g. \( 3x + 2x \geq 4 \). The various types of linear constraint are reflected in the following mathematical statement of the problem:

\[
\begin{align*}
\text{minimize } & F(x), \ x \text{ is in } \mathbb{R} \\
\text{subject to the } & \\
\text{equality } & a_{i} x = b_{i}, \ i=1,2,\ldots,m; \\
\text{constraints: } & i \\
\text{inequality } & a_{i} x \geq b_{i}, \ i=m+1,m+2,\ldots,m; \\
\text{constraints: } & i \ i \ 1 \ 1 \ 2 \\
\text{inequality } & a_{i} x \leq b_{i}, \ i=m+1,m+2,\ldots,m; \\
\text{constraints: } & i \ i \ 2 \ 2 \ 3 \\
\text{range } & s \leq a_{j} x \leq t_{j}, \ i=m+1,m+2,\ldots,m; \\
\text{constraints: } & j \ i \ j \ 3 \ 3 \ 4 \\
& j=1,2,\ldots,m-m; \\
& 4 \ 3 \\
\text{bounds } & 1 \leq x \leq u, \ i=1,2,\ldots,n
\end{align*}
\]
CHAPTER 15. CHAPTER N

constraints: \( a_i \) is a vector of length \( n \); \( b_i, s_i \) and \( t_i \) are constant scalars; and any of the categories may be empty.

Although the bounds on \( x_i \) could be included in the definition of general linear constraints, we prefer to distinguish between them for reasons of computational efficiency.

If \( F(x) \) is a linear function, the linearly-constrained problem is termed a linear programming problem (LP problem); if \( F(x) \) is a quadratic function, the problem is termed a quadratic programming problem (QP problem). For further discussion of LP and QP problems, including the dual formulation of such problems, see Dantzig [2].

2.1.5. Minimization subject to nonlinear constraints

A problem is included in this category if at least one constraint function is nonlinear, e.g. \( x_1^2 + x_3 + x_4 - 2 \geq 0 \). The mathematical statement of the problem is identical to that for the linearly-constrained case, except for the addition of the following constraints:

- equality \( c_i(x) = 0 \) \( i = 1,2,\ldots,m \);
- inequality \( c_i(x) \geq 0 \) \( i = m + 1, m + 2,\ldots,m \);
- range \( v \leq c_j(x) \leq w \) \( j = 1,2,\ldots,m \),

where each \( c_i \) is a nonlinear function; \( v \) and \( w \) are constant scalars; and any category may be empty. Note that we do not include a separate category for constraints of the form \( c_i(x) \leq 0 \), since this is equivalent to \( -c_i(x) \geq 0 \).

2.2. Geometric Representation and Terminology

To illustrate the nature of optimization problems it is useful to
consider the following example in two dimensions

\[
\begin{array}{ccc}
  x & 1 & 2 \\
  F(x) = e^{(4x + 2x + 4x_1 + 2x_2 + 1)}. & 2 & 2 \\
\end{array}
\]

(This function is used as the example function in the documentation for the unconstrained routines.)

Figure 1

Please see figure in printed Reference Manual

Figure 1 is a contour diagram of \( F(x) \). The contours labelled \( F_0, F_1, \ldots, F_4 \) are isovalue contours, or lines along which the function \( F(x) \) takes specific constant values. The point \( x \) is a local unconstrained minimum, that is, the value of \( F(x) \) is less than at all the neighbouring points. A function may have several such minima. The lowest of the local minima is termed a global minimum. In the problem illustrated in Figure 1, \( x \) is the only local minimum. The point \( x \) is said to be a saddle point because it is a minimum along the line \( AB \), but a maximum along \( CD \).

If we add the constraint \( x_1 \geq 0 \) to the problem of minimizing \( F(x) \), the solution remains unaltered. In Figure 1 this constraint is represented by the straight line passing through \( x_1 = 0 \), and the shading on the line indicates the unacceptable region. The region in \( \mathbb{R} \) satisfying the constraints of an optimization problem is termed the feasible region. A point satisfying the constraints is defined as a feasible point.

If we add the nonlinear constraint \( x_1 + x_2 - x_2^2 - x_1 - 1.5 \geq 0 \), represented by the curved shaded line in Figure 1, then \( x \) is not a feasible point. The solution of the new constrained problem is \( x \), the feasible point with the smallest function value.

2.2.1. Gradient vector
The vector of first partial derivatives of \( F(x) \) is called the gradient vector, and is denoted by \( g(x) \), i.e.,
\[
\begin{bmatrix}
\frac{dF(x)}{dx} & \frac{dF(x)}{dx} & \cdots & \frac{dF(x)}{dx}
\end{bmatrix}^T
\]
\( g(x) = [\cdots, \cdots, \cdots] \).
\[
= \begin{bmatrix}
\frac{dF(x)}{dx} & \frac{dF(x)}{dx} & \cdots & \frac{dF(x)}{dx}
\end{bmatrix}
\]
\[
= \begin{bmatrix}
1 & 2 & \cdots & n
\end{bmatrix}
\]

For the function illustrated in Figure 1,
\[
\begin{bmatrix}
x
1
\end{bmatrix}
\begin{bmatrix}
F(x) + e \cdot (8x + 4x)
\end{bmatrix}
\begin{bmatrix}
x
1
\end{bmatrix}
\begin{bmatrix}
1 & 2
\end{bmatrix}
\begin{bmatrix}
1 & 2
\end{bmatrix}
\begin{bmatrix}
e \cdot (4x + 4x + 2)
\end{bmatrix}
\begin{bmatrix}
2 & 1
\end{bmatrix}
\]
\[g(x) = \begin{bmatrix}
e \cdot (4x + 4x + 2)
\end{bmatrix}.
\begin{bmatrix}
2 & 1
\end{bmatrix}
\]

The gradient vector is of importance in optimization because it must be zero at an unconstrained minimum of any function with continuous first derivatives.

2.2.2. Hessian matrix

The matrix of second partial derivatives of a function is termed its Hessian matrix. The Hessian matrix of \( F(x) \) is denoted by \( G(x) \) and its \((i,j)\)th element is given by \( \frac{d^2 F(x)}{dx_i dx_j} \). If \( F(x) \)

has continuous second derivatives, then \( G(x) \) must be positive semi-definite at any unconstrained minimum of \( F \).

2.2.3. Jacobian matrix; matrix of constraint normals

In nonlinear least-squares problems, the matrix of first partial derivatives of the vector-valued function \( f(x) \) is termed the Jacobian matrix of \( f(x) \) and its \((i,j)\)th component is \( \frac{df_i}{dx_j} \).

The vector of first partial derivatives of the constraint \( c(x) \)

is denoted by
\[
\begin{bmatrix}
\frac{dc_i(x)}{dx} & \frac{dc_i(x)}{dx}
\end{bmatrix}^T
\]
\[a(x) = [\cdots, \cdots] \]
\[
= \begin{bmatrix}
\frac{dc_i(x)}{dx} & \frac{dc_i(x)}{dx}
\end{bmatrix}
\]
\[
= \begin{bmatrix}
\frac{dc_i(x)}{dx} & \frac{dc_i(x)}{dx}
\end{bmatrix}
\]
\[
= \begin{bmatrix}
1 & n
\end{bmatrix}
\]
At a point, $x$, the vector $a^i(x)$ is orthogonal (normal) to the
$i$
isovalue contour of $c^i(x)$ passing through $x$; this relationship is
illustrated for a two-dimensional function in Figure 2.

Figure 2
Please see figure in printed Reference Manual

The matrix whose columns are the vectors $\{a^i\}$ is termed the
$i$
matrix of constraint normals. Note that if $c^i(x)$ is a linear
$T$
constraint involving a $x^i$, then its vector of first partial
derivatives is simply the vector $a^i$.

2.3. Sufficient Conditions for a Solution

All nonlinear functions will be assumed to have continuous second
derivatives in the neighbourhood of the solution.

2.3.1. Unconstrained minimization

* The following conditions are sufficient for the point $x$ to be an
unconstrained local minimum of $F(x)$:

* (i) $||g(x)||=0$; and

* (ii) $G(x)$ is positive-definite,

where $||g||$ denotes the Euclidean length of $g$.

2.3.2. Minimization subject to bounds on the variables

At the solution of a bounds-constrained problem, variables which
are not on their bounds are termed free variables. If it is known
in advance which variables are on their bounds at the solution,
the problem can be solved as an unconstrained problem in just the
free variables; thus, the sufficient conditions for a solution
are similar to those for the unconstrained case, applied only to
the free variables.

* Sufficient conditions for a feasible point $x$ to be the solution of a bound-constrained problem are as follows:

* (i) $||g(x)||=0$; and

* (ii) $G(x)$ is positive-definite; and

* (i) $g(x)<0, x=u$; $g(x)>0, x=l$

where $g(x)$ is the gradient of $F(x)$ with respect to the free variables, and $G(x)$ is the Hessian matrix of $F(x)$ with respect to the free variables. The extra condition (iii) ensures that $F(x)$ cannot be reduced by moving off one or more of the bounds.

2.3.3. Linearly-constrained minimization

For the sake of simplicity, the following description does not include a specific treatment of bounds or range constraints, since the results for general linear inequality constraints can be applied directly to these cases.

At a solution $x$, of a linearly-constrained problem, the constraints which hold as equalities are called the active or binding constraints. Assume that there are $t$ active constraints at the solution $x$, and let $A$ denote the matrix whose columns are the columns of $A$ corresponding to the active constraints, with $b$ the vector similarly obtained from $b$; then

$$A^T * A x = b.$$

The matrix $Z$ is defined as an $n$ by $(n-t)$ matrix satisfying:

$$A^T T A Z = 0; \quad Z Z = I.$$
The columns of $Z$ form an orthogonal basis for the set of vectors orthogonal to the columns of $A$.

Define

$$T \begin{align*}
\mathbf{g}(x) &= Z g(x), \text{ the projected gradient vector of } F(x); \\
\begin{bmatrix} T \\
\end{bmatrix} 
\end{align*}$$

$$T \begin{align*}
\mathbf{G}(x) &= Z G(x)Z, \text{ the projected Hessian matrix of } F(x). \\
\end{align*}$$

At the solution of a linearly-constrained problem, the projected gradient vector must be zero, which implies that the gradient vector $g(x)$ can be written as a linear combination of the columns of $A$, i.e., $g(x) = \sum_{i} (\lambda_i) a_i = A(\lambda)$. The scalar $(\lambda_i)$ is defined as the Lagrange multiplier corresponding to the $i$th active constraint. A simple interpretation of the $i$th Lagrange multiplier is that it gives the gradient of $F(x)$ along the $i$th active constraint normal; a convenient definition of the Lagrange multiplier vector (although not a recommended method for computation) is:

$$^\top^\begin{bmatrix} T \\
\end{bmatrix} -1^\top^* \\
(\lambda) = (A A)^{-1} A g(x).$$

Sufficient conditions for $x$ to be the solution of a linearly-constrained problem are:

(i) $x$ is feasible, and $A x = b$; and

(ii) $\|g(x)\| = 0$, or equivalently, $g(x) = A(\lambda)$; and

(iii) $G(x)$ is positive-definite; and
(iv) \( (\lambda_i) > 0 \) if \( (\lambda_i) \) corresponds to a constraint
\[
\begin{align*}
^T a_i x & \geq b_i; \\
\end{align*}
\]

\( (\lambda_i) < 0 \) if \( (\lambda_i) \) corresponds to a constraint
\[
\begin{align*}
^T a_i x & \leq b_i. \\
\end{align*}
\]

The sign of \( (\lambda_i) \) is immaterial for equality constraints, which by definition are always active.

2.3.4. Nonlinearly-constrained minimization

For nonlinearly-constrained problems, much of the terminology is defined exactly as in the linearly-constrained case. The set of active constraints at \( x \) again means the set of constraints that hold as equalities at \( x \), with corresponding definitions of \( c \) and \( A \): the vector \( c(x) \) contains the active constraint functions, and the columns of \( A(x) \) are the gradient vectors of the active constraints. As before, \( Z \) is defined in terms of \( A(x) \) as a matrix such that:
\[
^T T \quad T \quad T \quad T \\
A Z = 0; \quad Z Z = I
\]

where the dependence on \( x \) has been suppressed for compactness.

The projected gradient vector \( g(x) \) is the vector \( Z g(x) \). At the solution \( x \) of a nonlinearly-constrained problem, the projected gradient must be zero, which implies the existence of Lagrange multipliers corresponding to the active constraints, i.e.,
\[
^T T \quad T \quad T \quad T \\
g(x) = A(x)(\lambda).
\]

The Lagrangian function is given by:
\[ L(x, \lambda) = F(x) - \lambda^T c(x). \]

We define \( g(x) \) as the gradient of the Lagrangian function; \( G(x) \)
\( L \)

as its Hessian matrix, and \( G^*(x) \) as its projected Hessian matrix,
\( L \)\( ^T \)

i.e., \( G^* = ZGZ \).
\( L \) \( L \) \( ^T \)

Sufficient conditions for \( x \) to be a solution of nonlinearly-constrained problem are:

(i) \( x \) is feasible, and \( c(x) = 0 \); and

\[ ||g(x)|| = 0, \text{ or equivalently, } g(x) = A(x)^T \lambda; \]

(ii) \( \lambda \) corresponds to a constraint of the form \( c_i \geq 0 \); the sign of \( \lambda \) is immaterial for an equality constraint.

Note that condition (ii) implies that the projected gradient of the Lagrangian function must also be zero at \( x \), since the application of \( Z \) annihilates the matrix \( A(x) \).

2.4. Background to Optimization Methods

All the algorithms contained in this Chapter generate an iterative sequence \( \{x_k\} \) that converges to the solution \( x \) in the limit, except for some special problem categories (i.e., linear and quadratic programming). To terminate computation of the sequence, a convergence test is performed to determine whether the current estimate of the solution is an adequate approximation. The convergence tests are discussed in Section 2.6.
Most of the methods construct a sequence \( \{x^{(k)} \} \) satisfying:

\[
x^{(k+1)} = x^{(k)} + (\alpha^{(k)}) p^{(k)},
\]

where the vector \( p^{(k)} \) is termed the direction of search, and \( (\alpha^{(k)}) \) is the steplength. The steplength \( (\alpha^{(k)}) \) is chosen so that \( F(x^{(k+1)}) < F(x^{(k)}) \).

2.4.1. Methods for unconstrained optimization

The distinctions among methods arise primarily from the need to use varying levels of information about derivatives of \( F(x) \) in defining the search direction. We describe three basic approaches to unconstrained problems, which may be extended to other problem categories. Since a full description of the methods would fill several volumes, the discussion here can do little more than allude to the processes involved, and direct the reader to other sources for a full explanation.

(a) **Newton-type Methods (Modified Newton Methods)**

Newton-type methods use the Hessian matrix \( G(x^{(k)}) \), or a finite difference approximation to \( G(x^{(k)}) \), to define the search direction. The routines in the Library either require a subroutine that computes the elements of \( G(x^{(k)}) \), or they approximate \( G(x^{(k)}) \) by finite differences.

Newton-type methods are the most powerful methods available for general problems and will find the minimum of a quadratic function in one iteration. See Sections 4.4. and 4.5.1. of Gill et al [5].

(b) **Quasi-Newton Methods**

Quasi-Newton methods approximate the Hessian \( G(x^{(k)}) \) by a matrix \( B^{(k)} \) which is modified at each iteration to include information obtained about the curvature of \( F \) along the latest search direction. Although not as robust as Newton-
type methods, quasi-Newton methods can be more efficient because G(x) is not computed, or approximated by finite-differences. Quasi-Newton methods minimize a quadratic function in n iterations. See Section 4.5.2 of Gill et al [5].

(c) Conjugate-Gradient Methods

Unlike Newton-type and quasi-Newton methods, conjugate gradient methods do not require the storage of an n by n matrix and so are ideally suited to solve large problems. Conjugate-gradient type methods are not usually as reliable or efficient as Newton-type, or quasi-Newton methods. See Section 4.8.3 of Gill et al [5].

2.4.2. Methods for nonlinear least-squares problems

These methods are similar to those for unconstrained optimization, but exploit the special structure of the Hessian matrix to give improved computational efficiency.

Since

\[
F(x) = \sum_{i=1}^{m} f(x)
\]

the Hessian matrix G(x) is of the form

\[
G(x)^T = 2\left[J(x) J(x)^T + \sum_{i=1}^{m} f(x) G_i(x)\right]
\]

where J(x) is the Jacobian matrix of f(x), and G_i(x) is the Hessian matrix of f_i(x).

In the neighbourhood of the solution, \(|||f(x)|||\) is often small compared to \(|||J(x) J(x)|||\) (for example, when f(x) represents the goodness of fit of a nonlinear model to observed data). In such cases, 2J(x) J(x) may be an adequate approximation to G(x),
thereby avoiding the need to compute or approximate second derivatives of $f(x)$. See Section 4.7 of Gill et al [5].

2.4.3. Methods for handling constraints

Bounds on the variables are dealt with by fixing some of the variables on their bounds and adjusting the remaining free variables to minimize the function. By examining estimates of the Lagrange multipliers it is possible to adjust the set of variables fixed on their bounds so that eventually the bounds active at the solution should be correctly identified. This type of method is called an active set method. One feature of such methods is that, given an initial feasible point, all $(k)$ approximations $x^{(k)}$ are feasible. This approach can be extended to general linear constraints. At a point, $x$, the set of constraints which hold as equalities being used to predict, or approximate, the set of active constraints is called the working set.

Nonlinear constraints are more difficult to handle. If at all possible, it is usually beneficial to avoid including nonlinear constraints during the formulation of the problem. The methods currently implemented in the Library handle nonlinearly constrained problems either by transforming them into a sequence of bound constraint problems, or by transforming them into a sequence of quadratic programming problems. A feature of almost all methods for nonlinear constraints is that $x^{(k)}$ is not guaranteed to be feasible except in the limit, and this is certainly true of the routines currently in the Library. See Chapter 6, particularly Section 6.4 and Section 6.5 of Gill et al [5].

Anyone interested in a detailed description of methods for optimization should consult the references.

2.5. Scaling

Scaling (in a broadly defined sense) often has a significant influence on the performance of optimization methods. Since convergence tolerances and other criteria are necessarily based on an implicit definition of 'small' and 'large', problems with unusual or unbalanced scaling may cause difficulties for some algorithms. Nonetheless, there are currently no scaling routines in the Library, although the position is under constant review. In light of the present state of the art, it is considered that sensible scaling by the user is likely to be more effective than any automatic routine. The following sections present some
general comments on problem scaling.

2.5.1. Transformation of variables

One method of scaling is to transform the variables from their original representation, which may reflect the physical nature of the problem, to variables that have certain desirable properties in terms of optimization. It is generally helpful for the following conditions to be satisfied:

(i) the variables are all of similar magnitude in the region of interest;

(ii) a fixed change in any of the variables results in similar changes in $F(x)$. Ideally, a unit change in any variable produces a unit change in $F(x)$;

(iii) the variables are transformed so as to avoid cancellation error in the evaluation of $F(x)$.

Normally, users should restrict themselves to linear transformations of variables, although occasionally nonlinear transformations are possible. The most common such transformation (and often the most appropriate) is of the form

$$x_{\text{new}} = D x_{\text{old}},$$

where $D$ is a diagonal matrix with constant coefficients. Our experience suggests that more use should be made of the transformation

$$x_{\text{new}} = D x_{\text{old}} + v,$$

where $v$ is a constant vector.

Consider, for example, a problem in which the variable $x_3$ represents the position of the peak of a Gaussian curve to be fitted to data for which the extreme values are 150 and 170; therefore $x_3$ is known to lie in the range 150--170. One possible scaling would be to define a new variable $x_3$, given by

$$x_3 = \frac{x_3 - 160}{2},$$
A better transformation, however, is given by defining $x$ as

$$x = \frac{x - 160}{10}.$$ 

Frequently, an improvement in the accuracy of evaluation of $F(x)$ can result if the variables are scaled before the routines to evaluate $F(x)$ are coded. For instance, in the above problem just mentioned of Gaussian curve fitting, $x$ may always occur in terms of the form $(x - x_m^3)$, where $x_m$ is a constant representing the mean peak position.

### 2.5.2. Scaling the objective function

The objective function has already been mentioned in the discussion of scaling the variables. The solution of a given problem is unaltered if $F(x)$ is multiplied by a positive constant, or if a constant value is added to $F(x)$. It is generally preferable for the objective function to be of the order of unity in the region of interest; thus, if in the original formulation $F(x)$ is always of the order of $10^{-5}$ (say), then the value of $F(x)$ should be multiplied by $10^5$ when evaluating the function within the optimization routines. If a constant is added or subtracted in the computation of $F(x)$, usually it should be omitted — i.e., it is better to formulate $F(x)$ as $x^2 + x$ rather than as $x^2 + x + 1000$ or even $x^2 + x + 1$. The inclusion of such a constant in the calculation of $F(x)$ can result in a loss of significant figures.

### 2.5.3. Scaling the constraints

The solution of a nonlinearly-constrained problem is unaltered if the $i$th constraint is multiplied by a positive weight $w_i$. At the approximation of the solution determined by a Library routine,
the active constraints will not be satisfied exactly, but will have 'small' values (for example, \( c_1 = 10^8 \), \( c_2 = 10^6 \), etc.). In general, this discrepancy will be minimized if the constraints are weighted so that a unit change in \( x \) produces a similar change in each constraint.

A second reason for introducing weights is related to the effect of the size of the constraints on the Lagrange multiplier estimates and, consequently, on the active set strategy. Additional discussion is given in Gill et al [5].

2.6. Analysis of Computed Results

2.6.1. Convergence criteria

The convergence criteria inevitably vary from routine to routine, since in some cases more information is available to be checked (for example, is the Hessian matrix positive-definite?), and different checks need to be made for different problem categories (for example, in constrained minimization it is necessary to verify whether a trial solution is feasible). Nonetheless, the underlying principles of the various criteria are the same; in non-mathematical terms, they are:

\[
\begin{align*}
(i) & \quad \text{is the sequence } \{x^{(k)}\} \text{ converging?} \\
(ii) & \quad \text{is the sequence } \{F^{(k)}\} \text{ converging?} \\
(iii) & \quad \text{are the necessary and sufficient conditions for the solution satisfied?}
\end{align*}
\]

The decision as to whether a sequence is converging is necessarily speculative. The criterion used in the present routines is to assume convergence if the relative change occurring between two successive iterations is less than some prescribed quantity. Criterion (iii) is the most reliable but often the conditions cannot be checked fully because not all the required information may be available.

2.6.2. Checking results

Little a priori guidance can be given as to the quality of the solution found by a nonlinear optimization algorithm, since no guarantees can be given that the methods will always work. Therefore, it is necessary for the user to check the computed solution even if the routine reports success. Frequently a
solution' may have been found even when the routine does not report a success. The reason for this apparent contradiction is that the routine needs to assess the accuracy of the solution. This assessment is not an exact process and consequently may be unduly pessimistic. Any 'solution' is in general only an approximation to the exact solution, and it is possible that the accuracy specified by the user is too stringent.

Further confirmation can be sought by trying to check whether or not convergence tests are almost satisfied, or whether or not some of the sufficient conditions are nearly satisfied. When it is thought that a routine has returned a non-zero value of IFAIL only because the requirements for 'success' were too stringent it may be worth restarting with increased convergence tolerances.

For nonlinearly-constrained problems, check whether the solution returned is feasible, or nearly feasible; if not, the solution returned is not an adequate solution.

Confidence in a solution may be increased by resolving the problem with a different initial approximation to the solution. See Section 8.3 of Gill et al [5] for further information.

2.6.3. Monitoring progress

Many of the routines in the Chapter have facilities to allow the user to monitor the progress of the minimization process, and users are encouraged to make use of these facilities. Monitoring information can be a great aid in assessing whether or not a satisfactory solution has been obtained, and in indicating difficulties in the minimization problem or in the routine’s ability to cope with the problem.

The behaviour of the function, the estimated solution and first derivatives can help in deciding whether a solution is acceptable and what to do in the event of a return with a non-zero value of IFAIL.

2.6.4. Confidence intervals for least-squares solutions

When estimates of the parameters in a nonlinear least-squares problem have been found, it may be necessary to estimate the variances of the parameters and the fitted function. These can be calculated from the Hessian of F(x) at the solution.

In many least-squares problems, the Hessian is adequately approximated at the solution by $G = 2J^T J$ (see Section 2.4.3). The Jacobian, J, or a factorization of J is returned by all the comprehensive least-squares routines and, in addition, a routine
is supplied in the Library to estimate variances of the parameters following the use of most of the nonlinear least-squares routines, in the case that \( G = 2J \) is an adequate approximation.

Let \( H \) be the inverse of \( G \), and \( S \) be the sum of squares, both calculated at the solution \( x \); an unbiased estimate of the variance of the \( i \)th parameter \( x \) is

\[
\text{var } x_i = \frac{2S}{m-n} H_{ii}
\]

and an unbiased estimate of the covariance of \( x_i \) and \( x_j \) is

\[
\text{covar}(x_i, x_j) = \frac{2S}{m-n} H_{ij}
\]

If \( x \) is the true solution, then the 100(1-(\(\beta\))) confidence interval on \( x \) is

\[
x_i - \frac{\text{var } x_i}{t} < x_i < x_i + \frac{\text{var } x_i}{t}, \quad i=1,2,...,n
\]

where \( t \) is the 100(1-(\(\beta\)))/2 percentage point of the t-distribution with \( m-n \) degrees of freedom.

In the majority of problems, the residuals \( f_i \), for \( i=1,2,...,m \),
contain the difference between the values of a model function 
\((\phi)(z,x)\) calculated for \(m\) different values of the independent 
variable \(z\), and the corresponding observed values at these 
points. The minimization process determines the parameters, or 

constants \(x\), of the fitted function \((\phi)(z,x)\). For any value, \(z\), 
of the independent variable \(z\), an unbiased estimate of the 
variance of \((\phi)\) is

\[
\text{var} (\phi) = \frac{2S}{m-n} \sum_{i=1}^{n} \sum_{j=1}^{n} \left( \frac{\partial^2 \phi}{\partial x_i \partial x_j} \right) \text{H}
\]

The 100(1-(\(\beta\))) confidence interval on \(F\) at the point \(z\) is

\[
(\phi)(z,x) - \sqrt{\text{var} (\phi)} t_{(\beta)/2,m-n} < (\phi)(z,x) + \sqrt{\text{var} (\phi)} t_{(\beta)/2,m-n}
\]

For further details on the analysis of least-squares solutions 
see Bard [1] and Wolberg [7].

2.7. References

Press.

Princeton University Press.

(2nd Edition).


3. Recommendations on Choice and Use of Routines

The choice of routine depends on several factors: the type of problem (unconstrained, etc.); the level of derivative information available (function values only, etc.); the experience of the user (there are easy-to-use versions of some routines); whether or not storage is a problem; and whether computational time has a high priority.

3.1. Choice of Routine

Routines are provided to solve the following types of problem:

- Nonlinear Programming
- Quadratic Programming
- Linear Programming
- Nonlinear Function (using 1st derivatives)
- Nonlinear Function, unconstrained or simple bounds (using function values only)
- Nonlinear least-squares (using function values only)
- Nonlinear least-squares (using function values and 1st derivatives)

E04UCF can be used to solve unconstrained, bound-constrained and linearly-constrained problems.

E04NAF can be used as a comprehensive linear programming solver; however, in most cases the easy-to-use routine E04MBF will be adequate.

E04MBF can be used to obtain a feasible point for a set of linear constraints.

E04DGF can be used to solve large scale unconstrained problems.

The routines can be used to solve problems in a single variable.

3.2. Service Routines

One of the most common errors in use of optimization routines is that the user’s subroutines incorrectly evaluate the relevant partial derivatives. Because exact gradient information normally
enhances efficiency in all areas of optimization, the user should be encouraged to provide analytical derivatives whenever possible. However, mistakes in the computation of derivatives can result in serious and obscure run-time errors, as well as complaints that the Library routines are incorrect.

E04UCF incorporates a check on the gradients being supplied and users are encouraged to utilize this option; E04GCF also incorporates a call to a derivative checker.

E04YCF estimates selected elements of the variance-covariance matrix for the computed regression parameters following the use of a nonlinear least-squares routine.

3.3. Function Evaluations at Infeasible Points

Users must not assume that the routines for constrained problems will require the objective function to be evaluated only at points which satisfy the constraints, i.e., feasible points. In the first place some of the easy-to-use routines call a service routine which will evaluate the objective function at the user-supplied initial point, and at neighbouring points (to check user-supplied derivatives or to estimate intervals for finite differencing). Apart from this, all routines will ensure that any evaluations of the objective function occur at points which approximately satisfy any simple bounds or linear constraints. Satisfaction of such constraints is only approximate because:

(a) routines which have a parameter FEATOL may allow such constraints to be violated by a margin specified by FEATOL;

(b) routines which estimate derivatives by finite differences may require function evaluations at points which just violate such constraints even though the current iteration just satisfies them.

There is no attempt to ensure that the current iteration satisfies any nonlinear constraints. Users who wish to prevent their objective function being evaluated outside some known region (where it may be undefined or not practically computable), may try to confine the iteration within this region by imposing suitable simple bounds or linear constraints (but beware as this may create new local minima where these constraints are active).

Note also that some routines allow the user-supplied routine to return a parameter (MODE) with a negative value to force an immediate clean exit from the minimization when the objective function cannot be evaluated.

3.4. Related Problems
Apart from the standard types of optimization problem, there are other related problems which can be solved by routines in this or other chapters of the Library.

E04MBF can be used to find a feasible point for a set of linear constraints and simple bounds.

Two routines in Chapter F04 solve linear least-squares problems,
\[ \minimize_{x} \sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} \]
where \( r(x) = \sum_{i}^{m} \sum_{j}^{n} a_{ij} x_{j} - b_{i} \).

E02GAF solves an overdetermined system of linear equations in the \( L_{1} \) norm, i.e., minimizes \( \sum_{i=1}^{m} |r_{i}(x)| \), with \( r \) as above.

E04 -- Minimizing or Maximizing a Function

Minimizing or Maximizing a Function

E04DGF Unconstrained minimum, pre-conditioned conjugate gradient algorithm, function of several variables using 1st derivatives

E04DJF Read optional parameter values for E04DGF from external file

E04DKF Supply optional parameter values to E04DGF

E04FDF Unconstrained minimum of a sum of squares, combined Gauss-Newton and modified Newton algorithm using function values only

E04GCF Unconstrained minimum of a sum of squares, combined Gauss-Newton and quasi-Newton algorithm, using 1st derivatives

E04JAF Minimum, function of several variables, quasi-Newton algorithm, simple bounds, using function values only

E04MBF Linear programming problem
CHAPTER 15. CHAPTER N

E04NAF  Quadratic programming problem
E04UCF  Minimum, function of several variables, sequential QP
         method, nonlinear constraints, using function values and
         optionally 1st derivatives
E04UDF  Read optional parameter values for E04UCF from external
         file
E04UEF  Supply optional parameter values to E04UCF
E04YCF  Covariance matrix for nonlinear least-squares problem

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function               E04DGF
      E04DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine
has been enhanced for use with AXIOM and is slightly different to
that offered in the standard version of the Foundation Library. In
particular, the optional parameters of the NAG routine are now
included in the parameter list. These are described in section
5.1.2, below.

1. Purpose

E04DGF minimizes an unconstrained nonlinear function of several
variables using a pre-conditioned, limited memory quasi-Newton
conjugate gradient method. First derivatives are required. The
routine is intended for use on large scale problems.

2. Specification

      SUBROUTINE E04DGF(N,OBJFUN,ITER,OBJF,OBJGRD,X,IWORK,WORK,IUSER,
      1   USER,ES,FU,IT,LIN,LIST,MA,OP,PR,STA,STO,
      2   VE,IFAIL)
      INTEGER N, ITER, IWORK(N+1), IUSER(*),
      1   IT, PR, STA, STO, VE, IFAIL
      DOUBLE PRECISION OBJF, OBJGRD(N), X(N), WORK(13*N), USER(*)
      LOGICAL LIST
      EXTERNAL OBJFUN

3. Description
E04DGF uses a pre-conditioned conjugate gradient method and is based upon algorithm PLMA as described in Gill and Murray [1] and Gill et al [2] Section 4.8.3.

The algorithm proceeds as follows:

Let $x$ be a given starting point and let $k$ denote the current iteration, starting with $k=0$. The iteration requires $g_k$, the gradient vector evaluated at $x_k$, the $k$th estimate of the minimum. At each iteration a vector $p_k$ (known as the direction of search) is computed and the new estimate $x_{k+1}$ is given by $x_k + (\alpha_k) p_k$ where $(\alpha_k)$ (the step length) minimizes the function $F(x_k + (\alpha_k) p_k)$ with respect to the scalar $(\alpha_k)$. A choice of initial step $(\alpha_0)$ is taken as

$$
(\alpha_0) = \min\{1, 2|F - F_{\text{est}}| / g_k^T g_k\}
$$

where $F_{\text{est}}$ is a user-supplied estimate of the function value at the solution. If $F_{\text{est}}$ is not specified, the software always chooses the unit step length for $(\alpha_k)$. Subsequent step length estimates are computed using cubic interpolation with safeguards.

A quasi-Newton method can be used to compute the search direction $p_k$ by updating the inverse of the approximate Hessian ($H_k$) and computing

$$
p_{k+1} = -H_k g_{k+1} \quad (1)
$$

The updating formula for the approximate inverse is given by

$$
H_k \leftarrow H_k - (T_{y^T y} + s_1 s_1^T) s s^T (I + \frac{1}{s_1^T y})
$$

$$
H_k = \frac{1}{1 + \frac{1}{s_1^T y}} (H_k + s_1 s_1^T)
$$

$$
H_k = \frac{1}{1 + \frac{1}{s_1^T y}} (H_k + s_1 s_1^T) \quad (2)
$$
where \( y = g - g \) and \( s = x - x = (\alpha) p \).

The method used by E04DGF to obtain the search direction is based upon computing \( p \) as \(-Hg\) where \( H \) is a matrix obtained by updating the identity matrix with a limited number of quasi-Newton corrections. The storage of an \( n \) by \( n \) matrix is avoided by storing only the vectors that define the rank two corrections — hence the term limited-memory quasi-Newton method. The precise method depends upon the number of updating vectors stored. For example, the direction obtained with the 'one-step' limited memory update is given by (1) using (2) with \( H \) equal to the identity matrix, viz.

\[
\begin{align*}
T & \quad (T) \\
sg & \quad (y y) \\
1 & \quad (T T) \\
\quad & \quad k+1 \quad k+1 \\
p & = -g + \frac{s y + y g s}{k+1} - \frac{s}{1 + \frac{s}{k+1}} \\
y & \quad (y s) \\
\quad & \quad k+1 \quad k \quad k+1 \\
s & \quad (y s) \\
\quad & \quad k \quad k \quad k
\end{align*}
\]

E04DGF uses a two-step method described in detail in Gill and Murray [1] in which restarts and pre-conditioning are incorporated. Using a limited-memory quasi-Newton formula, such as the one above, guarantees \( p \) to be a descent direction if all the inner products \( y \) are positive for all vectors \( y \) and \( s \) used in the updating formula.

The termination criterion of E04DGF is as follows:

Let \((\tau)\) specify a parameter that indicates the number of correct figures desired in \( F \) ((\tau) is equivalent to Optimality Tolerance in the optional parameter list, see Section 5.1). If the following three conditions are satisfied

(i) \( F - F < (\tau) (1 + |F|) \)

The termination criterion of E04DGF is as follows:

Let \((\tau)\) specify a parameter that indicates the number of correct figures desired in \( F \) ((\tau) is equivalent to Optimality Tolerance in the optional parameter list, see Section 5.1). If the following three conditions are satisfied

(i) \( F - F < (\tau) (1 + |F|) \)
(ii) \[ ||x - x^k|| < \frac{\tau}{(1+||x||)} \]

\[
\frac{\text{F}_{k-1}}{k-1} \]

(iii) \[ ||g|| \leq \frac{3}{(1+|F|)} \text{ or } ||g|| < \epsilon \]

where \( \epsilon \) is the absolute error associated with computing the objective function

then the algorithm is considered to have converged. For a full discussion on termination criteria see Gill et al [2] Chapter 8.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
On entry: the number \( n \) of variables. Constraint: \( N \geq 1 \).

2: \( \text{OBJFUN} \) -- SUBROUTINE, supplied by the user.

\textbf{External Procedure}

\texttt{OBJFUN} must calculate the objective function \( F(x) \) and its gradient for a specified \( n \) element vector \( x \).

Its specification is:

\[
\text{SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD,}
\]

\[
\text{NSTATE, IUSER, USER)}
\]

\[
\text{INTEGER MODE, N, NSTATE, IUSER(*)}
\]

\[
\text{DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)}
\]

1: \( \text{MODE} \) -- INTEGER Input/Output
\( \text{MODE} \) is a flag that the user may set within \( \text{OBJFUN} \) to indicate a failure in the evaluation of the objective function. On entry: \( \text{MODE} \) is always non-negative. On exit: if \( \text{MODE} \) is negative the execution of E04DGF is terminated with IFAIL set to \( \text{MODE} \).

2: \( N \) -- INTEGER Input
On entry: the number \( n \) of variables.

3: \( X(N) \) -- DOUBLE PRECISION array Input
On entry: the point x at which the objective function is required.

4: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function F at the current point x.

5: OBJGRD(N) -- DOUBLE PRECISION array Output
On exit: OBJGRD(i) must contain the value of \( \frac{\partial F}{\partial x_i} \) at the point x, for i=1,2,...,n.

6: NSTATE -- INTEGER Input
On entry: NSTATE will be 1 on the first call of OBJFUN by E04DGF, and is 0 for all subsequent calls. Thus, if the user wishes, NSTATE may be tested within OBJFUN in order to perform certain calculations once only. For example the user may read data or initialise COMMON blocks when NSTATE = 1.

7: IUSER(*) -- INTEGER array User Workspace
8: USER(*) -- DOUBLE PRECISION array User Workspace
OBJFUN is called from E04DGF with the parameters IUSER and USER as supplied to E04DGF. The user is free to use arrays IUSER and USER to supply information to OBJFUN as an alternative to using COMMON. OBJFUN must be declared as EXTERNAL in the (sub)program from which E04DGF is called. Parameters denoted as Input must not be changed by this procedure.

3: ITER -- INTEGER Output
On exit: the number of iterations performed.

4: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function F(x) at the final iterate.

5: OBJGRD(N) -- DOUBLE PRECISION array Output
On exit: the objective gradient at the final iterate.

6: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an initial estimate of the solution. On exit: the final estimate of the solution.

7: IWORK(N+1) -- INTEGER array Workspace

8: WORK(13*N) -- DOUBLE PRECISION array Workspace
9:  IUSER(*) -- INTEGER array  User Workspace
    Note: the dimension of the array IUSER must be at least 1.
    This array is not used by E04DGF, but is passed directly to
    routine OBJFUN and may be used to supply information to
    OBJFUN.

10: USER(*) -- DOUBLE PRECISION array  User Workspace
    Note: the dimension of the array USER must be at least 1.
    This array is not used by E04DGF, but is passed directly to
    routine OBJFUN and may be used to supply information to
    OBJFUN.

11: IFAIL -- INTEGER  Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).
    
    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit.

5.1. Optional Input Parameters

Several optional parameters in E04DGF define choices in the
behaviour of the routine. In order to reduce the number of formal
parameters of E04DGF these optional parameters have associated
default values (see Section 5.1.3) that are appropriate for most
problems. Therefore the user need only specify those optional
parameters whose values are to be different from their default
values.

The remainder of this section can be skipped by users who wish to
use the default values for all optional parameters. A complete
list of optional parameters and their default values is given in
Section 5.1.3.

5.1.1. Specification of the Optional Parameters

Optional parameters may be specified by calling one, or both, of
E04DJF and E04DKF prior to a call to E04DGF.

E04DJF reads options from an external options file, with Begin
and End as the first and last lines respectively and each
intermediate line defining a single optional parameter. For
example,

Begin
       Print Level = 1
End

The call

CALL E04DJF(IOPTNS, INFORM)

can then be used to read the file on unit IOPTNS. INFORM will be zero on successful exit. E04DJF should be consulted for a full description of this method of supplying optional parameters.

E04DKF can be called to supply options directly, one call being necessary for each optional parameter.

For example,

CALL E04DKF('Print level = 1')

E04DKF should be consulted for a full description of this method of supplying optional parameters.

All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user are unaltered by E04DGF (unless they define invalid values) and so remain in effect for subsequent calls to E04DGF, unless altered by the user.

5.1.2. Description of the Optional Parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and real values required with certain options. The number (epsilon) is a generic notation for machine precision, and (epsilon)_R denotes the relative precision of the objective function (the optional parameter Function Precision; see below).

Defaults

This special keyword may be used to reset the default values
following a call to E04DGF.

Estimated Optimal Function Value $r$

(Axiom parameter ES)

This value of $r$ specifies the user-supplied guess of the optimum objective function value. This value is used by E04DGF to calculate an initial step length (see Section 3). If the value of $r$ is not specified by the user (the default), then this has the effect of setting the initial step length to unity. It should be noted that for badly scaled functions a unit step along the steepest descent direction will often compute the function at very large values of $x$.

Function Precision $r$ Default = ($\epsilon_{\text{machine}}$)

(Axiom parameter FU)

The parameter defines ($\epsilon$), which is intended to be a measure of the accuracy with which the problem function $F$ can be computed. The value of ($\epsilon$) should reflect the relative precision of $1 + |F(x)|$; i.e. ($\epsilon$) acts as a relative precision when $|F|$ is large, and as an absolute precision when $|F|$ is small. For example, if $F(x)$ is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for ($\epsilon$) would be $1.0 \times 10^{-6}$. In contrast, if $F(x)$ is typically of order $10^{-4}$ and the first six significant digits are known to be correct, an appropriate value for ($\epsilon$) would be $1.0 \times 10^{-10}$. The choice of ($\epsilon$) can be quite complicated for badly scaled problems; see Chapter 8 of Gill and Murray [2], for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However when the accuracy of the computed function values is known to be significantly worse than full precision, the value of ($\epsilon$) should be large enough so that E04DGF will not attempt to distinguish between function values that differ by less than the error inherent in the calculation. If $0 < r < (\epsilon)$, where ($\epsilon$) is the machine precision then the default value is used.

Iteration Limit $i$ Default = $\max(50, 5n)$
Iters

Itns

(Axiom parameter IT)

The value \( i (i \geq 0) \) specifies the maximum number of iterations allowed before termination. If \( i < 0 \) the default value is used. See Section 8 for further information.

Linesearch Tolerance \( r \) Default = 0.9

(Axiom parameter LIN)

The value \( r (0 \leq r < 1) \) controls the accuracy with which the step (alpha) taken during each iteration approximates a minimum of the function along the search direction (the smaller the value of \( r \), the more accurate the linesearch). The default value \( r = 0.9 \) requests an inaccurate search, and is appropriate for most problems. A more accurate search may be appropriate when it is desirable to reduce the number of iterations - for example, if the objective function is cheap to evaluate.

List Default = List

Nolist

(Axiom parameter LIST)

Normally each optional parameter specification is printed as it is supplied. Nolist may be used to suppress the printing and List may be used to restore printing.

Maximum Step Length \( r \) Default = 10

(Axiom parameter MA)

The value \( r (r > 0) \) defines the maximum allowable step length for the line search. If \( r \leq 0 \) the default value is used.

Optimality Tolerance \( r \) Default = \((\text{epsilon})\)

(Axiom parameter OP)

\( R \)

The parameter \( r ((\text{epsilon}) \leq r < 1) \) specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, \( r \) indicates the number of correct
figures desired in the objective function at the solution. For
example, if \( r \) is 10 and E04DGF terminates successfully, the
final value of \( F \) should have approximately six correct figures.
E04DGF will terminate successfully if the iterative sequence of \( x \)
values is judged to have converged and the final point satisfies
the termination criteria (see Section 3, where (tau) represents
\( F \) Optimality Tolerance).

**Print Level i Default = 10**

(Axiom parameter PR)

The value \( i \) controls the amount of printout produced by E04DGF.
The following levels of printing are available.

- **0** No output.
- **1** The final solution.
- **5** One line of output for each iteration.
- **10** The final solution and one line of output for each
iteration.

**Start Objective Check at Variable i Default = 1**

(Axiom parameter STA)

**Stop Objective Check at Variable i Default = n**

(Axiom parameter STO)

These keywords take effect only if Verify Level > 0 (see below). They
may be used to control the verification of gradient elements computed by
subroutine OBJFUN. For example if the first 30 variables appear linearly in
the objective, so that the corresponding gradient elements are constant, then it is
reasonable to specify Start Objective Check at Variable 31.

**Verify Level i Default = 0**

- **Verify** No
- **Verify Level** -1
- **Verify Level** 0
Verify
Verify Yes
Verify Objective Gradients
Verify Gradients
Verify Level 1

(Axiom parameter VE)

These keywords refer to finite-difference checks on the gradient elements computed by the user-provided subroutine OBJFUN. It is possible to set Verify Level in several ways, as indicated above. For example, the gradients will be verified if Verify, Verify Yes, Verify Gradients, Verify Objective Gradients or Verify Level = 1 is specified.

If i<0 then no checking will be performed. If i>0 then the gradients will be verified at the user-supplied point. If i=0 only a 'cheap' test will be performed, requiring one call to OBJFUN. If i=1, a more reliable (but more expensive) check will be made on individual gradient components, within the ranges specified by the Start and Stop keywords as described above. A result of the form OK or BAD? is printed by E04DGF to indicate whether or not each component appears to be correct.

5.1.3. Optional parameter checklist and default values

For easy reference, the following sample list shows all valid keywords and their default values. The default options Function Precision and Optimality Tolerance depend upon (epsilon), the machine precision.

Optional Parameters Default Values

Estimated Optimal Function Value

0.9

Function precision (epsilon)

Iterations max(50,5n)

Linesearch Tolerance 0.9

10

Maximum Step Length 10
List/Nolist List

Optimality Tolerance 0.8

Print Level 10

Start Objective Check at 1 Variable

Stop Objective Check at n Variable

Verify Level 0

5.2. Description of Printed Output

The level of printed output from E04DGF is controlled by the user (see the description of Print Level in Section 5.1).

When Print Level >= 5, the following line of output is produced at each iteration.

Itn is the iteration count.

Step is the step (alpha) taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.

Nfun is the cumulated number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the gradients by finite differences are not included. Nfun is printed as a guide to the amount of work required for the linesearch. E04DGF will perform at most 16 function evaluations per iteration.

Objective is the value of the objective function.

Norm G is the Euclidean norm of the gradient of the objective function.

Norm X is the Euclidean norm of x.

Norm (X(k-1)-X(k)) is the Euclidean norm of x -x .

k-1  k

When Print Level = 1 or Print Level >= 10 then the solution at
the end of execution of E04DGF is printed out.

The following describes the printout for each variable:

Variable gives the name (VARBL) and index $j$ ($j = 1$ to $n$) of the variable

Value is the value of the variable at the final iterate

Gradient Value is the value of the gradient of the objective function with respect to the $j$th variable at the final iterate

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

On exit from E04DGF, IFAIL should be tested. If Print Level > 0 then a short description of IFAIL is printed.

Errors and diagnostics indicated by IFAIL from E04DGF are as follows:

IFAIL< 0
A negative value of IFAIL indicates an exit from E04DGF because the user set MODE negative in routine OBJFUN. The value of IFAIL will be the same as the user’s setting of MODE.

IFAIL= 1
Not used by this routine.

IFAIL= 2
Not used by this routine.

IFAIL= 3
The maximum number of iterations has been performed. If the algorithm appears to be making progress the iterations value may be too small (see Section 5.1.2) so the user should increase iterations and rerun E04DGF. If the algorithm seems to be 'bogged down', the user should check for incorrect gradients or ill-conditioning as described below under IFAIL = 6.

IFAIL= 4
The computed upper bound on the step length taken during the linesearch was too small. A rerun with an increased value of
the Maximum Step Length ((\rho) say) may be successful unless
10
(\rho) \geq 10 \quad (\text{the default value}), in which case the current
point cannot be improved upon.

IFAIL\ = \ 5
Not used by this routine.

IFAIL\ = \ 6
A sufficient decrease in the function value could not be
attained during the final linesearch. If the subroutine
OBJFUN computes the function and gradients correctly, then
this may occur because an overly stringent accuracy has been
requested, i.e., Optimality Tolerance is too small or if the
minimum lies close to a step length of zero. In this case
the user should apply the four tests described in Section 3
to determine whether or not the final solution is acceptable
(the user will need to set Print Level \geq 5). For a
discussion of attainable accuracy see Gill and Murray [2].

If many iterations have occurred in which essentially no
progress has been made or E04DGF has failed to move from the
initial point, subroutine OBJFUN may be incorrect. The user
should refer to the comments below under IFAIL = 7 and check
the gradients using the Verify parameter. Unfortunately,
there may be small errors in the objective gradients that
cannot be detected by the verification process. Finite-
difference approximations to first derivatives are
catastrophically affected by even small inaccuracies.

IFAIL\ = \ 7
Large errors were found in the derivatives of the objective
function. This value of IFAIL will occur if the verification
process indicated that at least one gradient component had
no correct figures. The user should refer to the printed
output to determine which elements are suspected to be in
error.

As a first step, the user should check that the code for the
objective values is correct - for example, by computing the
function at a point where the correct value is known.
However, care should be taken that the chosen point fully
tests the evaluation of the function. It is remarkable how
often the values x=0 or x=1 are used to test function
evaluation procedures, and how often the special properties
of these numbers make the test meaningless.

Special care should be used in this test if computation of
the objective function involves subsidiary data communicated
in COMMON storage. Although the first evaluation of the
function may be correct, subsequent calculations may be in error because some of the subsidiary data has accidentally been overwritten.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single-precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

**IFAIL= 8**
The gradient \( g \) at the starting point is too small. The value \( g g \) is less than \( (\text{epsilon}) |F(x)| \), where \( (\text{epsilon}) \) is the machine precision.

The problem should be rerun at a different starting point.

**IFAIL= 9**
On entry \( N < 1 \).

7. Accuracy

On successful exit the accuracy of the solution will be as defined by the optional parameter Optimality Tolerance.

8. Further Comments

Problems whose Hessian matrices at the solution contain sets of clustered eigenvalues are likely to be minimized in significantly fewer than \( n \) iterations. Problems without this property may require anything between \( n \) and \( 5n \) iterations, with approximately \( 2n \) iterations being a common figure for moderately difficult problems.

9. Example

To find a minimum of the function

\[
F = e^{(4x_1 + 2x_2 + 4x_1 x_2 + 2x_1 + 1)}. 
\]

\[
\begin{array}{cccc}
1 & 2 & 2 \\
1 & 2 & 1 & 2 & 2 \\
\end{array}
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function                      E04DJF
       E04DJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

To supply optional parameters to E04DGF from an external file.

2. Specification

    SUBROUTINE E04DJF (IOPTNS, INFORM)
        INTEGER IOPTNS, INFORM

3. Description

E04DJF may be used to supply values for optional parameters to E04DGF. E04DJF reads an external file and each line of the file defines a single optional parameter. It is only necessary to supply values for those parameters whose values are to be different from their default values.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

        Print level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran.
77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

The file containing the options must start with begin and must finish with end. An example of a valid options file is:

```
Begin * Example options file
Print level = 10
End
```

Normally each line of the file is printed as it is read, on the current advisory message unit (see X04ABF), but printing may be suppressed using the keyword nolist. To suppress printing of begin, nolist must be the first option supplied as in the file:

```
Begin
Nolist
Print level = 10
End
```

Printing will automatically be turned on again after a call to E04DGF and may be turned on again at any time by the user by using the keyword list.

Optional parameter settings are preserved following a call to E04DGF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the routine document for E04DGF.

4. References

None.

5. Parameters

1: IOPTNS -- INTEGER
   Input
   On entry: IOPTNS must be the unit number of the options file. Constraint: 0 <= IOPTNS <= 99.

2: INFORM -- INTEGER
   Output
   On exit: INFORM will be zero if an options file with the
correct structure has been read. Otherwise INFORM will be positive. Positive values of INFORM indicate that an options file may not have been successfully read as follows:

\[ \text{INFORM} = 1 \]

\[ \text{IOPTNS is not in the range } [0,99]. \]

\[ \text{INFORM} = 2 \]

\[ \text{begin was found, but end-of-file was found before end was found.} \]

\[ \text{INFORM} = 3 \]

\[ \text{end-of-file was found before begin was found.} \]

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning message is output on the current advisory message unit (see X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04DKF may also be used to supply optional parameters to E04DGF.

9. Example

See the example for E04DGF.
3. Description

E04DKF may be used to supply values for optional parameters to E04DGF. It is only necessary to call E04DKF for those parameters whose values are to be different from their default values. One call to E04DKF sets one parameter value.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

```
Print Level = 1
```

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist. Thus the statement

```
CALL E04DKF ('Nolist')
```

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04DGF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to E04DGF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

```
CALL E04DKF ('Defaults')
```
prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the routine document for E04DGF.

4. References
None.

5. Parameters

1: STRING -- CHARACTER(*)
   Input
   On entry: STRING must be a single valid option string. See Section 3 above, and Section 5.1 of the routine document for E04DGF.

6. Error Indicators and Warnings
If the parameter STRING is not recognised as a valid option string, then a warning message is output on the current advisory message unit (see X04ABF).

7. Accuracy
Not applicable.

8. Further Comments
E04DJF may also be used to supply optional parameters to E04DGF.

9. Example
See the example for E04DGF.

E04 -- Minimizing or Maximizing a Function
E04FDF
E04FDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
E04FDF is an easy-to-use algorithm for finding an unconstrained minimum of a sum of squares of m nonlinear functions in n
variables (m>=n). No derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```fortran
SUBROUTINE E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
INTEGER M, N, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION X(N), FSUMSQ, W(LW)
```

3. Description

This routine is essentially identical to the subroutine LSNDN1 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

\[
\text{Minimize } F(x) = \sum_{i=1}^{m} f_i(x)^2
\]

where \(x=(x_1, x_2, \ldots, x_n)\) and \(m\geq n\). (The functions \(f_i(x)\) are often referred to as 'residuals'.) The user must supply a subroutine LSFUN1 to evaluate functions \(f_i(x)\) at any point \(x\).

From a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of the sum of squares. These points are generated using estimates of the curvature of \(F(x)\).

4. References


5. Parameters

1:  \(M \cdots \text{INTEGER} \quad \text{Input} \)

2:  \(N \cdots \text{INTEGER} \quad \text{Input} \)

On entry: the number \(m\) of residuals \(f_i(x)\), and the number \(n\) of variables, \(x\). Constraint: \(1 \leq N \leq M\).
3: X(N) -- DOUBLE PRECISION array  Input/Output
   On entry: X(j) must be set to a guess at the jth component
   of the position of the minimum, for j=1,2,...,n. On exit:
   the lowest point found during the calculations. Thus, if
   IFAIL = 0 on exit, X(j) is the jth component of the position
   of the minimum.

4: FSUMSQ -- DOUBLE PRECISION  Output
   On exit: the value of the sum of squares, F(x),
   corresponding to the final point stored in X.

5: IW(LIW) -- INTEGER array  Workspace

6: LIW -- INTEGER  Input
   On entry: the length of IW as declared in the (sub)program
   from which E04FDF has been called. Constraint: LIW >= 1.

7: W(LW) -- DOUBLE PRECISION array  Workspace

8: LW -- INTEGER  Input
   On entry: the length of W as declared in the (sub)program
   from which E04FDF is called. Constraints:
   LW >= N*(7 + N + 2*M + (N-1)/2) + 3*M, if N > 1,
   LW >= 9 + 5>M, if N = 1.

9: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. Users who are
   unfamiliar with this parameter should refer to the Essential
   Introduction for details.

   On exit: IFAIL = 0 unless the routine detects an error or
   gives a warning (see Section 6).

   For this routine, because the values of output parameters
   may be useful even if IFAIL /=0 on exit, users are
   recommended to set IFAIL to -1 before entry. It is then
   essential to test the value of IFAIL on exit.

5.1. Optional Parameters

   LSFUN1 -- SUBROUTINE, supplied by the user.
   External Procedure
   This routine must be supplied by the user to calculate the
   vector of values f (x) at any point x. Since the routine is
   not a parameter to E04FDF, it must be called LSFUN1. It
   should be tested separately before being used in conjunction
with E04FDF (see the Chapter Introduction).

Its specification is:

```plaintext
SUBROUTINE LSFUN1 (M, N, XC, FVECC)
INTEGER M, N
DOUBLE PRECISION XC(N), FVECC(M)

1: M -- INTEGER  
   Input
2: N -- INTEGER  
   On entry: the numbers m and n of residuals and
   variables, respectively.
3: XC(N) -- DOUBLE PRECISION array  
   On entry: the point x at which the values of the f
   i
   are required.
4: FVECC(M) -- DOUBLE PRECISION array  
   On exit: FVECC(i) must contain the value of f at the
   i
   point x, for i=1,2,...,m.

LSFUN1 must be declared as EXTERNAL in the (sub)program
from which E04FDF is called. Parameters denoted as
Input must not be changed by this procedure.
```

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry N < 1,

or M < N,

or LIW < 1,

or LW < N*(7 + N + 2*M + (N-1)/2) + 3*M, when N > 1,

or LW < 9 + 5*M, when N = 1.

IFAIL= 2
There have been 400*n calls of LSFUN1, yet the algorithm
does not seem to have converged. This may be due to an
awkward function or to a poor starting point, so it is worth
restarting E04FDF from the final point held in X.
IFAIL= 3
The final point does not satisfy the conditions for acceptance as a minimum, but no lower point could be found.

IFAIL= 4
An auxiliary routine has been unable to complete a singular value decomposition in a reasonable number of sub-iterations.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
There is some doubt about whether the point $x$ found by E04FDF is a minimum of $F(x)$. The degree of confidence in the result decreases as IFAIL increases. Thus when IFAIL = 5, it is probable that the final $x$ gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

If the user is not satisfied with the result (e.g. because IFAIL lies between 3 and 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made, then, for a computer with a mantissa of $t$ decimals, one would expect to get about $t/2 - 1$ decimals accuracy in the components of $x$ and between $t - 1$ (if $F(x)$ is of order 1 at the minimum) and $2t - 2$ (if $F(x)$ is close to zero at the minimum) decimals accuracy in $F(x)$.

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the distance of the starting point from the solution. The number of multiplications performed per iteration of E04FDF varies, but for $m \gg n$ is approximately $n^2m + O(n^2)$. In addition, each iteration makes at least $n+1$ calls of LSFUN1. So, unless the residuals can be evaluated very quickly, the run time will be dominated by the
time spent in LSFUN1.

Ideally, the problem should be scaled so that the minimum value of the sum of squares is in the range (0,1), and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04FDF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array W. See E04YCF for further details.

9. Example

To find least-squares estimates of $x_1$, $x_2$ and $x_3$ in the model

$$y = x_1 + \frac{1}{x_2 + x_3}$$

using the 15 sets of data given in the following table.

<table>
<thead>
<tr>
<th>y</th>
<th>t 1</th>
<th>t 2</th>
<th>t 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td>1.0</td>
<td>15.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.18</td>
<td>2.0</td>
<td>14.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.22</td>
<td>3.0</td>
<td>13.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.25</td>
<td>4.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.29</td>
<td>5.0</td>
<td>11.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.32</td>
<td>6.0</td>
<td>10.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.35</td>
<td>7.0</td>
<td>9.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.39</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>0.37</td>
<td>9.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.58</td>
<td>10.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.73</td>
<td>11.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.96</td>
<td>12.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>1.34</td>
<td>13.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.10</td>
<td>14.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>4.39</td>
<td>15.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The program uses (0.5, 1.0, 1.5) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

### E04GCF

**E04GCF** -- NAG Foundation Library Routine Document

**Note:** Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

E04GCF is an easy-to-use quasi-Newton algorithm for finding an unconstrained minimum of a sum of squares of \( m \) nonlinear functions in \( n \) variables (\( m \geq n \)). First derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. **Specification**

```fortran
SUBROUTINE E04GCF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
INTEGER M, N, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION X(N), FSUMSQ, W(LW)
```

3. **Description**

This routine is essentially identical to the subroutine LSFDQ2 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

\[
\begin{align*}
\text{minimize } & \quad \sum_{i=1}^{m} [f_i(x)]^2 \\
\text{subject to } & \quad x \in \mathbb{R}^n
\end{align*}
\]

where \( x=(x_1, x_2, \ldots, x_n) \) and \( m \geq n \). (The functions \( f_i(x) \) are often referred to as 'residuals'.) The user must supply a subroutine LSFUN2 to evaluate the residuals and their first derivatives at
any point $x$.

Before attempting to minimize the sum of squares, the algorithm checks LSFUN2 for consistency. Then, from a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of the sum of squares. These points are generated using estimates of the curvature of $F(x)$.

4. References


5. Parameters

1: M -- INTEGER Input

2: N -- INTEGER Input
   On entry: the number $m$ of residuals $f_i(x)$, and the number $n$ of variables, $x_j$. Constraint: $1 \leq N \leq M$.

3: X(N) -- DOUBLE PRECISION array Input/Output
   On entry: $X(j)$ must be set to a guess at the $j$th component of the position of the minimum, for $j=1,2,\ldots,n$. The routine checks the first derivatives calculated by LSFUN2 at the starting point, and so is more likely to detect an error in the user's routine if the initial $X(j)$ are non-zero and mutually distinct. On exit: the lowest point found during the calculations. Thus, if IFAIL = 0 on exit, $X(j)$ is the $j$th component of the position of the minimum.

4: FSUMSQ -- DOUBLE PRECISION Output
   On exit: the value of the sum of squares, $F(x)$, corresponding to the final point stored in $X$.

5: IW(LIW) -- INTEGER array Workspace

6: LIW -- INTEGER Input
   On entry: the length of IW as declared in the (sub)program from which E04GCF is called. Constraint: LIW $\geq 1$.

7: W(LW) -- DOUBLE PRECISION array Workspace

8: LW -- INTEGER Input
   On entry: the length of W as declared in the (sub)program from which E04GCF is called. Constraints:
LW ≥ 2*N*(4 + N + M) + 3*M, if N > 1,

LW ≥ 11 + 5*M, if N = 1.

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

5.1. Optional Parameters

LSFUN2 -- SUBROUTINE, supplied by the user.

External Procedure

This routine must be supplied by the user to calculate the vector of values \( f(x) \) and the Jacobian matrix of first \( df_i \) \( ddx_j \) derivatives ---- at any point \( x \). Since the routine is not a parameter to E04GCF, it must be called LSFUN2. It should be tested separately before being used in conjunction with E04GCF (see the Chapter Introduction).

Its specification is:

```fortran
SUBROUTINE LSFUN2 (M, N, XC, FVECC, FJACC, LJC)
INTEGER M, N, LJC
DOUBLE PRECISION XC(N), FVECC(M), FJACC(LJC,N)
```

Important: The dimension declaration for FJACC must contain the variable LJC, not an integer constant.

1: M -- INTEGER Input

2: N -- INTEGER Input
On entry: the numbers m and n of residuals and variables, respectively.

3: XC(N) -- DOUBLE PRECISION array Input
On entry: the point \( x \) at which the values of the \( f_i \)
ddf
  i
and the ---- are required.
  ddx
  j

4:  FVECC(M) -- DOUBLE PRECISION array Output
On exit: FVECC(i) must contain the value of f at the
  i
point x, for i=1,2,...,m.

5:  FJACC(LJC,N) -- DOUBLE PRECISION array Output
On exit: FJACC(i,j) must contain the value of ---- at
  ddf
ddx
  i
  j
the point x, for i=1,2,...,m; j=1,2,...,n.

6:  LJC -- INTEGER Input
On entry: the first dimension of the array FJACC.
LSFUN2 must be declared as EXTERNAL in the (sub)program
from which E04GCF is called. Parameters denoted as
Input must not be changed by this procedure.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=1
  On entry N < 1,
  or M < N,
  or LIW < 1,
  or LW < 2*N*(4 + N + M) + 3*M, when N > 1,
  or LW < 9 + 5*M, when N = 1.

IFAIL=2
  There have been 50*n calls of LSFUN2, yet the algorithm does
not seem to have converged. This may be due to an awkward
function or to a poor starting point, so it is worth
restarting E04GCF from the final point held in X.

IFAIL=3
The final point does not satisfy the conditions for acceptance as a minimum, but no lower point could be found.

IFAIL= 4
An auxiliary routine has been unable to complete a singular value decomposition in a reasonable number of sub-iterations.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
There is some doubt about whether the point X found by E04GCF is a minimum of F(x). The degree of confidence in the result decreases as IFAIL increases. Thus, when IFAIL = 5, it is probable that the final x gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

IFAIL= 9
It is very likely that the user has made an error in forming
\begin{align*}
\text{ddf}^i \\
\text{dx}_j
\end{align*}
the derivatives \text{----} in LSFUN2.

If the user is not satisfied with the result (e.g. because IFAIL lies between 3 and 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made then, for a computer with a mantissa of t decimals, one would expect to get t/2-1 decimals accuracy in the components of x and between t-1 (if F(x) is of order 1 at the minimum) and 2t-2 (if F(x) is close to zero at the minimum) decimals accuracy in F(x).

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the
distance of the starting point from the solution. The number of multiplications performed per iteration of E04GCF varies, but for \( m \gg n \) is approximately \( n^m + O(n) \). In addition, each iteration makes at least one call of LSFUN2. So, unless the residuals and their derivatives can be evaluated very quickly, the run time will be dominated by the time spent in LSFUN2.

Ideally the problem should be scaled so that the minimum value of the sum of squares is in the range \((0,1)\) and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04GCF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array \( W \). See E04YCF for further details.

9. Example

To find the least-squares estimates of \( x_1 \), \( x_2 \) and \( x_3 \) in the model

\[
y = x_1 + \frac{1}{x_1 x_2 + x_3}
\]

using the 15 sets of data given in the following table.

<table>
<thead>
<tr>
<th>( y )</th>
<th>( t )</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td>1.0</td>
<td>15.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0.18</td>
<td>2.0</td>
<td>14.0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>0.22</td>
<td>3.0</td>
<td>13.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>4.0</td>
<td>12.0</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>0.29</td>
<td>5.0</td>
<td>11.0</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>0.32</td>
<td>6.0</td>
<td>10.0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>0.35</td>
<td>7.0</td>
<td>9.0</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>0.39</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>0.37</td>
<td>9.0</td>
<td>7.0</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>0.58</td>
<td>10.0</td>
<td>6.0</td>
<td>6.0</td>
<td></td>
</tr>
</tbody>
</table>
The program uses (0.5, 1.0, 1.5) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
when derivatives of \( F(x) \) are unavailable.

Special provision is made for problems which actually have no bounds on the \( x \), problems which have only non-negativity bounds \( j \) and problems in which \( 1 = l_1 = \ldots = l_n \) and \( u = u_1 = \ldots = u_n \). The user must supply a subroutine \( \text{FUNCT}1 \) to calculate the value of \( F(x) \) at any point \( x \).

From a starting point supplied by the user there is generated, on the basis of estimates of the gradient and the curvature of \( F(x) \), a sequence of feasible points which is intended to converge to a local minimum of the constrained function. An attempt is made to verify that the final point is a minimum.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
   On entry: the number \( n \) of independent variables.
   Constraint: \( N \geq 1 \).

2: \( IBOUND \) -- INTEGER Input
   On entry: indicates whether the facility for dealing with bounds of special forms is to be used.

   It must be set to one of the following values:
   \( IBOUND = 0 \)
   if the user will be supplying all the \( l_j \) and \( u_j \) individually.

   \( IBOUND = 1 \)
   if there are no bounds on any \( x_j \).

   \( IBOUND = 2 \)
   if all the bounds are of the form \( 0 \leq x_j \).

   \( IBOUND = 3 \)
   if \( 1 = l_1 = \ldots = l_n \) and \( u = u_1 = \ldots = u_n \).
3: BL(N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the lower bounds $l^j$.
   
   If IBOUND is set to 0, the user must set $BL(j)$ to $l^j$, for
   
   $j=1,2,...,n$. (If a lower bound is not specified for a
   
   particular $x$, the corresponding $BL(j)$ should be set to -10.)
   
   If IBOUND is set to 3, the user must set $BL(1)$ to $l^1$; E04JAF
   will then set the remaining elements of $BL$ equal to $BL(1)$.
   On exit: the lower bounds actually used by E04JAF.

4: BU(N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the upper bounds $u^j$.
   
   If IBOUND is set to 0, the user must set $BU(j)$ to $u^j$, for
   
   $j=1,2,...,n$. (If an upper bound is not specified for a
   
   particular $x$, the corresponding $BU(j)$ should be set to 10.)
   
   If IBOUND is set to 3, the user must set $BU(1)$ to $u^1$; E04JAF
   will then set the remaining elements of $BU$ equal to $BU(1)$.
   On exit: the upper bounds actually used by E04JAF.

5: X(N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: $X(j)$ must be set to an estimate of the $j$th
   component of the position of the minimum, for $j=1,2,...,n$.
   On exit: the lowest point found during the calculations.
   Thus, if IFAIL = 0 on exit, $X(j)$ is the $j$th component of the
   position of the minimum.

6: F -- DOUBLE PRECISION  
   Output
   On exit: the value of $F(x)$ corresponding to the final point
   stored in $X$.

7: IW(LIW) -- INTEGER array  
   Workspace

8: LIW -- INTEGER  
   Input
   On entry: the length of $IW$ as declared in the (sub)program
   from which E04JAF is called. Constraint: $LIW >= N + 2$.

9: W(LW) -- DOUBLE PRECISION array  
   Workspace
CHAPTER 15. CHAPTER N

10: LW -- INTEGER
     On entry: the length of W as declared in the (sub)program
     from which E04JAF is called. Constraint: LW\geq\max(N\times(N-1)/2+12\times N,13).

11: IFAIL -- INTEGER
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

5.1. Optional Parameters

FUNCT1 -- SUBROUTINE, supplied by the user.

     External Procedure

     This routine must be supplied by the user to calculate the
     value of the function F(x) at any point x. Since this
     routine is not a parameter to E04JAF, it must be called
     FUNCT1. It should be tested separately before being used in
     conjunction with E04JAF (see the Chapter Introduction).

     Its specification is:

     SUBROUTINE FUNCT1 (N, XC, FC)
     INTEGER N
     DOUBLE PRECISION XC(N), FC

     1: N -- INTEGER
        On entry: the number n of variables.

     2: XC(N) -- DOUBLE PRECISION array
        On entry: the point x at which the function value is
        required.

     3: FC -- DOUBLE PRECISION
        On exit: the value of the function F at the current
        point x.

     FUNCT1 must be declared as EXTERNAL in the (sub)program
     from which E04JAF is called. Parameters denoted as
     Input must not be changed by this procedure.
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
On entry N < 1,
or IBOUND < 0,
or IBOUND > 3,
or IBOUND = 0 and BL(j) > BU(j) for some j,
or IBOUND = 3 and BL(1) > BU(1),
or LIW < N + 2,
or LW<max(13,12\ast N+N\ast(N-1)/2).

IFAIL= 2
There have been 400\ast n function evaluations, yet the algorithm does not seem to be converging. The calculations can be restarted from the final point held in X. The error may also indicate that F(x) has no minimum.

IFAIL= 3
The conditions for a minimum have not all been met but a lower point could not be found and the algorithm has failed.

IFAIL= 4
An overflow has occurred during the computation. This is an unlikely failure, but if it occurs the user should restart at the latest point given in X.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
There is some doubt about whether the point x found by E04JAF is a minimum. The degree of confidence in the result decreases as IFAIL increases. Thus, when IFAIL = 5 it is probable that the final x gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

IFAIL= 9
In the search for a minimum, the modulus of one of the 6 variables has become very large ($\sim 10^6$). This indicates that there is a mistake in FUNCT1, that the user's problem has no finite solution, or that the problem needs rescaling (see Section 8).

If the user is dissatisfied with the result (e.g. because IFAIL = 5, 6, 7 or 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. If persistent trouble occurs and the gradient can be calculated, it may be advisable to change to a routine which uses gradients (see the Chapter Introduction).

7. Accuracy

When a successful exit is made then, for a computer with a mantissa of $t$ decimals, one would expect to get about $t/2-1$ decimals accuracy in $x$ and about $t-1$ decimals accuracy in $F$, provided the problem is reasonably well scaled.

8. Further Comments

The number of iterations required depends on the number of variables, the behaviour of $F(x)$ and the distance of the starting point from the solution. The number of operations performed in an iteration of E04JAF is roughly proportional to $n$. In addition, each iteration makes at least $m+1$ calls of FUNCT1, where $m$ is the number of variables not fixed on bounds. So, unless $F(x)$ can be evaluated very quickly, the run time will be dominated by the time spent in FUNCT1.

Ideally the problem should be scaled so that at the solution the value of $F(x)$ and the corresponding values of $x_1, x_2, \ldots, x_n$ are each in the range $(-1,+1)$, and so that at points a unit distance away from the solution, $F$ is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04JAF will take less computer time.

9. Example

To minimize

$$2^2 2^4 4^4$$
\[ F = (x_1 + 10x_2) + 5(x_1 - x_2) + (x_2 - 2x_3) + 10(x_4 - x_1) \]

subject to

\[
\begin{align*}
1 &\leq x &\leq 3 \\
1 &\leq x &\leq 3
\end{align*}
\]

starting from the initial guess \((3, -1, 0, 1)\).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**E04MBF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

E04MBF is an easy-to-use routine for solving linear programming problems, or for finding a feasible point for such problems. It is not intended for large sparse problems.

2. **Specification**

```fortran
SUBROUTINE E04MBF (ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 A, BL, BU, CVEC, LINOBJ, X, ISTATE,
2 OBJLP, CLAMDA, IWORK, LIWORK, WORK,
3 LWORK, IFAIL)
INTEGER ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 ISTATE(NCTOTL), IWORK(LIWORK), LIWORK,
2 LWORK, IFAIL
DOUBLE PRECISION A(NROWA,N), BL(NCTOTL), BU(NCTOTL), CVEC
1 (N), X(N), OBJLP, CLAMDA(NCTOTL), WORK
2 (LWORK)
LOGICAL LINOBJ
```
3. Description

E04MBF solves linear programming (LP) problems of the form

\[
\begin{align*}
T (x) & \\
\text{Minimize} & \quad c^T x \quad \text{subject to} \quad l \leq (Ax) \leq u \quad (LP) \\
\text{n} & \\
x \text{ is in } \mathbb{R}^n
\end{align*}
\]

where \(c\) is an \(n\) element vector and \(A\) is an \(m \times n\) matrix i.e.,
there are \(n\) variables and \(m\) general linear constraints. \(m\) may be
zero in which case the LP problem is subject only to bounds on
the variables. Notice that upper and lower bounds are specified
for all the variables and constraints. This form allows full
generality in specifying other types of constraints. For example
the \(i\)th constraint may be specified as equality by setting \(l = u\).

If certain bounds are not present the associated elements of \(l\) or
\(u\) can be set to special values that will be treated as \(-\infty\) or
\(+\infty\).

The routine allows the linear objective function to be omitted in
which case a feasible point for the set of constraints is sought.

The user must supply an initial estimate of the solution.

Users who wish to exercise additional control and users with
problems whose solution would benefit from additional flexibility
should consider using the comprehensive routine E04NAF.

4. References


User’s Guide for SOL/QPSOL. Report SOL 83-7. Department of
Operations Research, Stanford University.

5. Parameters

1: ITMAX -- INTEGER  
   Input
   On entry: an upper bound on the number of iterations to be
taken. If ITMAX is not positive, then the value 50 is used
in place of ITMAX.

2: MSGLVL -- INTEGER  
   Input
   On entry: indicates whether or not printout is required at
the final solution. When printing occurs the output is on
the advisory message channel (see X04ABF). A description of
the printed output is given in Section 5.1. The level of printing is determined as follows:

MSGLVL < 0
No printing.

MSGLVL = 0
Printing only if an input parameter is incorrect, or if the problem is so ill-conditioned that subsequent overflow is likely. This setting is strongly recommended in preference to MSGLVL < 0.

MSGLVL = 1
Printing at the solution.

MSGLVL > 1
Values greater than 1 should normally be used only at the direction of NAG; such values may generate large amounts of printed output.

3: N -- INTEGER Input

4: NCLIN -- INTEGER Input
   On entry: the number of general linear constraints in the problem. Constraint: NCLIN >= 0.

5: NCTOTL -- INTEGER Input
   On entry: the value (N+NCLIN).

6: NROWA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which E04MBF is called.
   Constraint: NROWA >= max(1,NCLIN).

7: A(NROWA,N) -- DOUBLE PRECISION array Input
   On entry: the leading NCLIN by n part of A must contain the NCLIN general constraints, with the coefficients of the ith constraint in the ith row of A. If NCLIN = 0, then A is not referenced.

8: BL(NCTOTL) -- DOUBLE PRECISION array Input
   On entry: the first n elements of BL must contain the lower bounds on the n variables, and when NCLIN > 0, the next NCLIN elements of BL must contain the lower bounds on the NCLIN general linear constraints. To specify a non-existent lower bound (1 = -infty), set BL(j)<=-1.0E+20.

9: BU(NCTOTL) -- DOUBLE PRECISION array Input
On entry: the first $n$ elements of $BU$ must contain the upper bounds on the $n$ variables, and when $NCLIN > 0$, the next $NCLIN$ elements of $BU$ must contain the upper bounds on the $NCLIN$ general linear constraints. To specify a non-existent upper bound ($u = +\infty$), set $BU(j) = 1.0E+20$. Constraint: 

\[
BL(j) \leq BU(j), \quad \text{for } j = 1, 2, \ldots, NCTOTL.
\]

10: $CVEC(N)$ -- DOUBLE PRECISION array
On entry: with $LINOBJ = .TRUE.$, $CVEC$ must contain the coefficients of the objective function. If $LINOBJ = .FALSE.$, then $CVEC$ is not referenced.

11: $LINOBJ$ -- LOGICAL
On entry: indicates whether or not a linear objective function is present. If $LINOBJ = .TRUE.$, then the full LP problem is solved, but if $LINOBJ = .FALSE.$, only a feasible point is found and the array $CVEC$ is not referenced.

12: $X(N)$ -- DOUBLE PRECISION array
On entry: an estimate of the solution, or of a feasible point. Even when $LINOBJ = .TRUE.$, it is not necessary for the point supplied in $X$ to be feasible. In the absence of better information all elements of $X$ may be set to zero. On exit: the solution to the LP problem when $LINOBJ = .TRUE.$, or a feasible point when $LINOBJ = .FALSE.$.

When no feasible point exists (see $IFAIL = 1$ in Section 6) then $X$ contains the point for which the sum of the infeasibilities is a minimum. On return with $IFAIL = 2, 3$ or $4$, $X$ contains the point at which E04MBF terminated.

13: $ISTATE(NCTOTL)$ -- INTEGER array
On exit: with $IFAIL < 5$, $ISTATE$ indicates the status of every constraint at the final point. The first $n$ elements of $ISTATE$ refer to the upper and lower bounds on the variables and when $NCLIN > 0$ the next $NCLIN$ elements refer to the general constraints.

Their meaning is:

\[
ISTATE(j) \quad \text{Meaning}
\]

-2 The constraint violates its lower bound. This value cannot occur for any element of $ISTATE$ when a feasible point has been found.

-1 The constraint violates its upper bound. This value cannot occur for any element of $ISTATE$ when a feasible point has been found.
The constraint is not in the working set (is not active) at the final point. Usually this means that the constraint lies strictly between its bounds.

This inequality constraint is in the working set (is active) at its lower bound.

This inequality constraint is in the working set (is active) at its upper bound.

This constraint is included in the working set (is active) as an equality. This value can only occur when BL(j) = BU(j).

On exit: when LINOBJ = .TRUE., then on successful exit, OBJLP contains the value of the objective function at the solution, and on exit with IFAIL = 2, 3 or 4, OBJLP contains the value of the objective function at the point returned in X.

When LINOBJ = .FALSE., then on successful exit OBJLP will be zero and on return with IFAIL = 1, OBJLP contains the minimum sum of the infeasibilities corresponding to the point returned in X.

On exit: when LINOBJ = .TRUE., then on successful exit, or on exit with IFAIL = 2, 3, or 4, CLAMDA contains the Lagrange multipliers (reduced costs) for each constraint with respect to the working set. The first n components of CLAMDA contain the multipliers for the bound constraints on the variables and the remaining NCLIN components contain the multipliers for the general linear constraints.

If ISTATE(j) = 0 so that the jth constraint is not in the working set then CLAMDA(j) is zero. If X is optimal and ISTATE(j) = 1, then CLAMDA(j) should be non-negative, and if ISTATE(j) = 2, then CLAMDA(j) should be non-positive.

When LINOBJ = .FALSE., all NCTOTL elements of CLAMDA are returned as zero.

On entry: the length of the array IWORK as declared in the (sub)program from which E04MBF is called. Constraint: LIWORK>=2*N.
2202

CHAPTER 15. CHAPTER N

18: WORK(LWORK) -- DOUBLE PRECISION array
Workspace

19: LWORK -- INTEGER
Input
On entry: the length of the array WORK as declared in the
(sub)program from which E04MBF is called. Constraints:
when \( N \leq \text{NCLIN} \) then
\[
2 \times \text{LWORK} \geq 2 \times N + 6 \times N + 4 \times \text{NCLIN} + \text{NROWA};
\]
when \( 0 \leq \text{NCLIN} < N \) then
\[
2 \times \text{LWORK} \geq 2 \times (\text{NCLIN} + 1) + 4 \times \text{NCLIN} + 6 \times N + \text{NROWA}.
\]

20: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
the output of an error message when soft failure occurs, set
IFAIL to 1.

5.1. Description of the Printed Output

When MSGLVL = 1, then E04MBF will produce output on the advisory
message channel (see X04ABF ), giving information on the final
point. The following describes the printout associated with each
variable.

Output Meaning

VARBL The name (V) and index j, for j=1,2,...,n, of the
variable.

STATE The state of the variable. (FR if neither bound is
in the working set, EQ for a fixed variable, LL if
on its lower bound, UL if on its upper bound and
TB if held on a temporary bound.) If the value of
the variable lies outside the upper or lower bound
then STATE will be ++ or -- respectively.

VALUE The value of the variable at the final iteration.
LOWER BOUND  The lower bound specified for the variable.
UPPER BOUND  The upper bound specified for the variable.
LAGR MULT   The value of the Lagrange multiplier for the associated bound.
RESIDUAL    The difference between the value of the variable and the nearer of its bounds.

For each of the general constraints the printout is as above with refers to the jth element of Ax, except that VARBL is replaced by:

LNCON       The name (L) and index j, for j = 1,2,...,NCLIN of the constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

Note: when MSGLVL=1 a short description of the error is printed.

IFAIL= 1
No feasible point could be found. Moving violated constraints so that they are satisfied at the point returned in X gives the minimum moves necessary to make the LP problem feasible.

IFAIL= 2
The solution to the LP problem is unbounded.

IFAIL= 3
A total of 50 changes were made to the working set without altering x. Cycling is probably occurring. The user should consider using E04NAF with MSGLVL >= 5 to monitor constraint additions and deletions in order to determine whether or not cycling is taking place.

IFAIL= 4
The limit on the number of iterations has been reached. Increase ITMAX or consider using E04NAF to monitor progress.

IFAIL= 5
An input parameter is invalid. Unless MSGLVL < 0 a message will be printed.

IFAIL=Overflow
If the printed output before the overflow occurred contains a warning about serious ill-conditioning in the working set
when adding the jth constraint, then either the user should try using E04NAF and experiment with the magnitude of FEATOL (j) in that routine, or the offending linearly dependent constraint (with index j) should be removed from the problem.

7. Accuracy

The routine implements a numerically stable active set strategy and returns solutions that are as accurate as the condition of the LP problem warrants on the machine.

8. Further Comments

The time taken by each iteration is approximately proportional to \(2^{\min(n,\text{NCLIN})}\).

Sensible scaling of the problem is likely to reduce the number of iterations required and make the problem less sensitive to perturbations in the data, thus improving the condition of the LP problem. In the absence of better information it is usually sensible to make the Euclidean lengths of each constraint of comparable magnitude. See Gill et al [1] for further information and advice.

Note that the routine allows constraints to be violated by an absolute tolerance equal to the machine precision (see X02AJF(*)).

9. Example

To minimize the function

\[-0.02x_{1} -0.2x_{2} -0.2x_{3} -0.2x_{4} -0.2x_{5} +0.04x_{6} +0.04x_{7}\]

subject to the bounds

\[-0.01 \leq x_{1} \leq 0.01,\]
\[-0.1 \leq x_{2} \leq 0.15,\]
\[-0.01 \leq x_{3} \leq 0.03,\]
\[-0.04 \leq x_{4} \leq 0.02,\]
\[-0.1 \leq x_{5} \leq 0.05,\]
\[-0.01 \leq x_{6} \leq 0.01.\]
-0.01 <= x
  7

and the general constraints

\[
\begin{align*}
  x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 & = -0.13 \\
  0.15x_1 + 0.04x_2 + 0.02x_3 + 0.04x_4 + 0.02x_5 + 0.01x_6 + 0.03x_7 & <= -0.0049 \\
  0.03x_1 + 0.05x_2 + 0.08x_3 + 0.02x_4 + 0.06x_5 + 0.01x_6 & <= -0.0064 \\
  0.02x_1 + 0.04x_2 + 0.01x_3 + 0.02x_4 + 0.02x_5 & <= -0.0037 \\
  0.02x_1 + 0.03x_2 + 0.01x_5 & <= -0.0012 \\
  -0.0992 <= 0.70x_1 + 0.75x_2 + 0.80x_3 + 0.75x_4 + 0.80x_5 + 0.97x_6 \\
  -0.003 <= 0.02x_1 + 0.06x_2 + 0.08x_3 + 0.12x_4 + 0.02x_5 + 0.01x_6 + 0.97x_7 <= 0.002
\end{align*}
\]

The initial point, which is infeasible, is

\[ x = (-0.01, -0.03, 0.0, -0.01, -0.1, 0.02, 0.01) . \]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function

E04NAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04NAF is a comprehensive routine for solving quadratic programming (QP) or linear programming (LP) problems. It is not
intended for large sparse problems.

2. Specification

SUBROUTINE E04NAF (ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 NROWH, NCOLH, BIGBND, A, BL, BU, CVEC,
2 FEATOL, HESS, QPHESS, COLD, LP, ORTHOG,
3 X, ISTATE, ITER, OBJ, CLAMDA, IWORK,
4 LIWORK, WORK, LWORK, IFAIL)

INTEGER ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 NROWH, NCOLH, ISTATE(NCTOTL), ITER, IWORK
2 (LIWORK), LIWORK, LWORK, IFAIL

DOUBLE PRECISION BIGBND, A(NROWA,N), BL(NCTOTL),
1 BU(NCTOTL), CVEC(N), FEATOL(NCTOTL), HESS
2 (NROWH,NCOLH), X(N), OBJ, CLAMDA(NCTOTL),
3 WORK(LWORK)

LOGICAL COLD, LP, ORTHOG

EXTERNAL QPHESS

3. Description

E04NAF is essentially identical to the subroutine SOL/QPSOL described in Gill et al [1].

E04NAF is designed to solve the quadratic programming (QP) problem - the minimization of a quadratic function subject to a set of linear constraints on the variables. The problem is assumed to be stated in the following form:

\[
\begin{align*}
T & \quad T \\
\text{Minimize} & \quad c^T x + \frac{1}{2} x^T H x \\
\text{subject to} & \quad l \leq (Ax) \leq u,
\end{align*}
\]

where \( c \) is a constant \( n \)-vector and \( H \) is a constant \( n \times n \) symmetric matrix; note that \( H \) is the Hessian matrix (matrix of second partial derivatives) of the quadratic objective function. The matrix \( A \) is \( m \times n \), where \( m \) may be zero; \( A \) is treated as a dense matrix.

The constraints involving \( A \) will be called the general constraints. Note that upper and lower bounds are specified for all the variables and for all the general constraints. The form of (1) allows full generality in specifying other types of constraints. In particular, an equality constraint is specified by setting \( l_i = u_i \). If certain bounds are not present, the associated elements of \( l \) or \( u \) can be set to special values that will be treated as \(-\infty\) or \(+\infty\).

The user must supply an initial estimate of the solution to (1),
and a subroutine that computes the product $Hx$ for any given vector $x$. If $H$ is positive-definite or positive semi-definite, E04NAF will obtain a global minimum; otherwise, the solution obtained will be a local minimum (which may or may not be a global minimum). If $H$ is defined as the zero matrix, E04NAF will solve the resulting linear programming (LP) problem; however, this can be accomplished more efficiently by setting a logical variable in the call of the routine (see the parameter LP in Section 5).

E04NAF allows the user to provide the indices of the constraints that are believed to be exactly satisfied at the solution. This facility, known as a warm start, can lead to significant savings in computational effort when solving a sequence of related problems.

The method has two distinct phases. In the first (the LP phase), an iterative procedure is carried out to determine a feasible point. In this context, feasibility is defined by a user-provided array FEATOL; the $j$th constraint is considered satisfied if its violation does not exceed FEATOL$(j)$. The second phase (the QP phase) generates a sequence of feasible iterates in order to minimize the quadratic objective function. In both phases, a subset of the constraints - called the working set - is used to define the search direction at each iteration; typically, the working set includes constraints that are satisfied to within the corresponding tolerances in the FEATOL array.

We now briefly describe a typical iteration in the QP phase. Let $x$ denote the estimate of the solution at the $k$th iteration; the next iterate is defined by

$$x_{k+1} = x_k + (\alpha_k) p_k$$

where $p_k$ is an $n$-dimensional search direction and $(\alpha_k)$ is a scalar step length. Assume that the working (active) set contains $t$ linearly independent constraints, and let $C_k$ denote the matrix of coefficients of the bounds and general constraints in the current working set.

Let $Z_k$ denote a matrix whose columns form a basis for the null space of $C_k$, so that $C_k Z_k = 0$. (Note that $Z_k$ has $n$ columns, where $n = n-t$.) The vector $Z_k (c + H_k x_k)$ is called the projected gradient
at $x$. If the projected gradient is zero at $x$ (i.e., $x$ is a
constrained stationary point in the subspace defined by $Z$),
Lagrange multipliers ($\lambda$) are defined as the solution of the
compatible overdetermined system

$$ T\lambda = c + Hx $$

The Lagrange multiplier ($\lambda$) corresponding to an inequality
constraint in the working set is said to be optimal if
($\lambda$)$\leq 0$ when the associated constraint is at its upper bound,
or if ($\lambda$)$\geq 0$ when the associated constraint is at its lower
bound. If a multiplier is non-optimal, the objective function can
be reduced by deleting the corresponding constraint (with index
JDEL, see Section 5.1) from the working set.

If the projected gradient at $x$ is non-zero, the search direction
$p$ is defined as

$$ p = Zp $$

where $p$ is an $n$-vector. In effect, the constraints in the
working set are treated as equalities, by constraining $p$ to lie
within the subspace of vectors orthogonal to the rows of $C$. This
definition ensures that $Cp = 0$, and hence the values of the
constraints in the working set are not altered by any move along
$p$.

The vector $p$ is obtained by solving the equations

$$ T Z HZ p = - Z (c + Hx) $$

(The matrix $Z HZ$ is called the projected Hessian matrix.) If the
If the projected Hessian is positive-definite and \( x + p \) is feasible, \((\alpha)\) will be taken as unity. In this case, the projected gradient at \( x \) will be zero (see NORM ZTG in Section 5.1), and Lagrange multipliers can be computed (see Gill et al [2]). Otherwise, \((\alpha)\) is set to the step to the 'nearest' \( k \) constraint (with index JADD, see Section 5.1), which is added to the working set at the next iteration.

The matrix \( Z \) is obtained from the TQ factorization of \( C \), in which \( C \) is represented as

\[
C Q = (0 \ T)
\]

where \( T \) is reverse-triangular. It follows from (5) that \( Z \) may be taken as the first \( n \) columns of \( Q \). If the projected Hessian is positive-definite, (3) is solved using the Cholesky factorization

\[
T \ T
Z HZ = R \ R
\]

where \( R \) is upper triangular. These factorizations are updated as \( k \) constraints enter or leave the working set (see Gill et al [2] for further details).

An important feature of E04NAF is the treatment of indefiniteness in the projected Hessian. If the projected Hessian is positive-definite, it may become indefinite only when a constraint is deleted from the working set. In this case, a temporary modification (of magnitude HESS MOD, see Section 5.1) is added to the last diagonal element of the Cholesky factor. Once a modification has occurred, no further constraints are deleted from the working set until enough constraints have been added so...
that the projected Hessian is again positive-definite. If equation (1) has a finite solution, a move along the direction obtained by solving (4) with the modified Cholesky factor must encounter a constraint that is not already in the working set.

In order to resolve indefiniteness in this way, we must ensure that the projected Hessian is positive-definite at the first iterate in the QP phase. Given the \( n \times n \) projected Hessian, a step-wise Cholesky factorization is performed with symmetric interchanges (and corresponding rearrangement of the columns of \( Z \)), terminating if the next step would cause the matrix to become indefinite. This determines the largest possible positive-definite principal sub-matrix of the (permuted) projected Hessian. If \( n \) steps of the Cholesky factorization have been successfully completed, the relevant projected Hessian is an \( n \times n \) positive-definite matrix \( Z^T H Z \), where \( Z \) comprises the first \( n \) columns of \( Z \). The quadratic function will subsequently be minimized within subspaces of reduced dimension until the full projected Hessian is positive-definite.

If a linear program is being solved and there are fewer general constraints than variables, the method moves from one vertex to another while minimizing the objective function. When necessary, an initial vertex is defined by temporarily fixing some of the variables at their initial values.

Several strategies are used to control ill-conditioning in the working set. One such strategy is associated with the \texttt{FEATOL} array. Allowing the \( j \)th constraint to be violated by as much as \( \texttt{FEATOL} (j) \) often provides a choice of constraints that could be added to the working set. When a choice exists, the decision is based on the conditioning of the working set. Negative steps are occasionally permitted, since \( x_k \) may violate the constraint to be added.

4. References


5. Parameters

1: \textbf{ITMAX} -- INTEGER \hspace{1cm} \textbf{Input}
   
   On entry: an upper bound on the number of iterations to be taken during the LP phase or the QP phase. If ITMAX is not positive, then the value 50 is used in place of ITMAX.

2: \textbf{MSGLVL} -- INTEGER \hspace{1cm} \textbf{Input}
   
   On entry: MSGLVL must indicate the amount of intermediate output desired (see Section 5.1 for a description of the printed output). All output is written to the current advisory message unit (see X04ABF). For MSGLVL $\geq 10$, each level includes the printout for all lower levels.
   
   Value Definition
   
   $< 0$ No printing.
   
   0 Printing only if an input parameter is incorrect, or if the working set is so ill-conditioned that subsequent overflow is likely. This setting is strongly recommended in preference to MSGLVL $< 0$.
   
   1 The final solution only.
   
   5 One brief line of output for each constraint addition or deletion (no printout of the final solution).
   
   $\geq 10$ The final solution and one brief line of output for each constraint addition or deletion.
   
   $\geq 15$ At each iteration, $X$, ISTATE, and the indices of the free variables (i.e., the variables not currently held on a bound).
   
   $\geq 20$ At each iteration, the Lagrange multiplier estimates and the general constraint values.
   
   $\geq 30$ At each iteration, the diagonal elements of the matrix $T$ associated with the $TQ$ factorization of the working set, and the diagonal elements of the Cholesky factor $R$ of the projected Hessian.
   
   $\geq 80$ Debug printout.
The arrays CVEC and HESS.

3: N -- INTEGER
   Input
   On entry: the number, n, of variables. Constraint: N \geq 1.

4: NCLIN -- INTEGER
   Input
   On entry: the number of general linear constraints in the
   problem. Constraint: NCLIN \geq 0.

5: NCTOTL -- INTEGER
   Input
   On entry: the value (N+NCLIN).

6: NROWA -- INTEGER
   Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which E04NAF is called.
   Constraint: NROWA \geq \max(1,NCLIN).

7: NROWH -- INTEGER
   Input
   On entry: the first dimension of the array HESS as declared
   in the (sub)program from which E04NAF is called.
   Constraint: NROWH \geq 1.

8: NCOLH -- INTEGER
   Input
   On entry: the column dimension of the array HESS as declared
   in the (sub)program from which E04NAF is called.
   Constraint: NCOLH \geq 1.

9: BIGBND -- DOUBLE PRECISION
   Input
   On entry: BIGBND must denote an 'infinite' component of l
   and u. Any upper bound greater than or equal to BIGBND will
   be regarded as plus infinity, and a lower bound less than or
   equal to -BIGBND will be regarded as minus infinity.
   Constraint: BIGBND > 0.0.

10: A(NROWA,N) -- DOUBLE PRECISION array
    Input
    On entry: the leading NCLIN by n part of A must contain the
    NCLIN general constraints, with the ith constraint in the i
    th row of A. If NCLIN = 0, then A is not referenced.

11: BL(NCTOTL) -- DOUBLE PRECISION array
    Input
    On entry: the lower bounds for all the constraints, in the
    following order. The first n elements of BL must contain the
    lower bounds on the variables. If NCLIN \geq 0, the next NCLIN
    elements of BL must contain the lower bounds for the general
    linear constraints. To specify a non-existent lower bound
    (i.e., 1 =-infty), the value used must satisfy BL(j)\leq-
    j
    BIGBND To specify the jth constraint as an equality, the
user must set BL(j) = BU(j). Constraint: BL(j) <= BU(j), j=1,2,...,NCTOTL.

12: BU(NCTOTL) -- DOUBLE PRECISION array Input
On entry: the upper bounds for all the constraints, in the following order. The first n elements of BU must contain the upper bounds on the variables. If NCLIN > 0, the next NCLIN elements of BU must contain the upper bounds for the general linear constraints. To specify a non-existent upper bound (i.e., u = +infty), the value used must satisfy BU(j) >= j
BIGBND. To specify the jth constraint as an equality, the user must set BU(j) = BL(j). Constraint: BU(j) >= BL(j), j=1,2,...,NCTOTL.

13: CVEC(N) -- DOUBLE PRECISION array Input
On entry: the coefficients of the linear term of the objective function (the vector c in equation (1)).

14: FEATOL(NCTOTL) -- DOUBLE PRECISION array Input
On entry: a set of positive tolerances that define the maximum permissible absolute violation in each constraint in order for a point to be considered feasible, i.e., if the violation in constraint j is less than FEATOL(j), the point is considered to be feasible with respect to the jth constraint. The ordering of the elements of FEATOL is the same as that described above for BL.

The elements of FEATOL should not be too small and a warning message will be printed on the current advisory message channel if any element of FEATOL is less than the machine precision (see X02AJF(*)). As the elements of FEATOL increase, the algorithm is less likely to encounter difficulties with ill-conditioning and degeneracy. However, larger values of FEATOL(j) mean that constraint j could be violated by a significant amount. It is recommended that FEATOL(j) be set to a value equal to the largest acceptable violation for constraint j. For example, if the data defining the constraints are of order unity and are correct to about 6 decimal digits, it would be appropriate to choose -6 FEATOL(j) as 10^-6 for all relevant j. Often the square root of the machine precision is a reasonable choice if the constraint is well scaled.

15: HESS(NROWH,NCOLH) -- DOUBLE PRECISION array Input
On entry: HESS may be used to store the Hessian matrix H of equation (1) if desired. HESS is accessed only by the subroutine QPHESS and is not accessed if LP = .TRUE.. Refer to the specification of QPHESS (below) for further details.
of how HESS may be used to pass data to QPHESS.

16: QPHESS -- SUBROUTINE, supplied by the user.

External Procedure

QPHESS must define the product of the Hessian matrix $H$ and a vector $x$. The elements of $H$ need not be defined explicitly. QPHESS is not accessed if LP is set to .TRUE. and in this case QPHESS may be the dummy routine E04NAN. (E04NAN is included in the NAG Foundation Library and so need not be supplied by the user. Its name may be implementation-dependent: see the Users’ Note for your implementation for details.)

Its specification is:

```fortran
SUBROUTINE QPHESS (N, NROWH, NCOLH, JTHCOL,
                   HESS, X, HX)
   INTEGER N, NROWH, NCOLH, JTHCOL
   DOUBLE PRECISION HESS(NROWH,NCOLH), X(N), HX(N)
```

1: $N$ -- INTEGER
   Input
   On entry: the number $n$ of variables.

2: $NROWH$ -- INTEGER
   Input
   On entry: the row dimension of the array HESS.

3: $NCOLH$ -- INTEGER
   Input
   On entry: the column dimension of the array HESS.

4: $JTHCOL$ -- INTEGER
   Input
   The input parameter $JTHCOL$ is included to allow flexibility for the user in the special situation when $x$ is the $j$th co-ordinate vector (i.e., the $j$th column of the identity matrix). This may be of interest because the product $Hx$ is then the $j$th column of $H$, which can sometimes be computed very efficiently. The user may code QPHESS to take advantage of this case. On entry: if $JTHCOL = j$, where $j>0$, $HX$ must contain column $JTHCOL$ of $H$, and hence special code may be included in QPHESS to test $JTHCOL$ if desired. However, special code is not necessary, since the vector $x$ always contains column $JTHCOL$ of the identity matrix whenever QPHESS is called with $JTHCOL > 0$.

5: $HESS(NROWH,NCOLH)$ -- DOUBLE PRECISION array
   Input
   On entry: the Hessian matrix $H$.

In some cases, it may be desirable to use a one-dimensional array to transmit data or workspace to QPHESS; $HESS$ should then be declared with dimension
In other situations, it may be desirable to compute \( Hx \) without accessing \( HESS \) - for example, if \( H \) is sparse or has special structure. (This is illustrated in the subroutine \texttt{QPHES1} in the example program in Section 9.) The parameters \( HESS, NROWH \) and \( NCOLH \) may then refer to any convenient array.

When \texttt{MSGLVL} = 99, the (possibly undefined) contents of \( HESS \) will be printed, except if \( NROWH \) and \( NCOLH \) are both 1. Also printed are the results of calling \texttt{QPHESS} with \( JTHCOL = 1,2,\ldots,n \).

6: \textbf{X(N)} -- DOUBLE PRECISION array \hspace{1cm} Input
On entry: the vector \( x \).

7: \textbf{HX(N)} -- DOUBLE PRECISION array \hspace{1cm} Output
On exit: \( HX \) must contain the product \( Hx \).

\texttt{QPHESS} must be declared as \texttt{EXTERNAL} in the (sub)program from which \texttt{E04NAF} is called. Parameters denoted as Input must not be changed by this procedure.

17: \textbf{COLD} -- LOGICAL \hspace{1cm} Input
On entry: \texttt{COLD} must indicate whether the user has specified an initial estimate of the active set of constraints. If \texttt{COLD} is set to \texttt{.TRUE.}, the initial working set is determined by \texttt{E04NAF}. If \texttt{COLD} is set to \texttt{.FALSE.} (a 'warm start'), the user must define the \texttt{ISTATE} array which gives the status of each constraint with respect to the working set. \texttt{E04NAF} will override the user's specification of \texttt{ISTATE} if necessary, so that a poor choice of working set will not cause a fatal error.

The warm start option is particularly useful when \texttt{E04NAF} is called repeatedly to solve related problems.

18: \textbf{LP} -- LOGICAL \hspace{1cm} Input
On entry: if \texttt{LP} = \texttt{.FALSE.}, \texttt{E04NAF} will solve the specified quadratic programming problem. If \texttt{LP} = \texttt{.TRUE.}, \texttt{E04NAF} will treat \( H \) as zero and solve the resulting linear programming problem; in this case, the parameters \( HESS \) and \texttt{QPHESS} will not be referenced.

19: \textbf{ORTHOG} -- LOGICAL \hspace{1cm} Input
On entry: \texttt{ORTHOG} must indicate whether orthogonal transformations are to be used in computing and updating the TQ factorization of the working set
\[ A Q = (0 \ T), \]
where $A$ is a sub-matrix of $A$ and $T$ is reverse-triangular.

If ORTHOG = .TRUE., the TQ factorization is computed using Householder reflections and plane rotations, and the matrix $Q$ is orthogonal. If ORTHOG = .FALSE., stabilized elementary transformations are used to maintain the factorization, and $Q$ is not orthogonal. A rule of thumb in making the choice is that orthogonal transformations require more work, but provide greater numerical stability. Thus, we recommend setting ORTHOG to .TRUE. if the problem is reasonably small or the active set is ill-conditioned. Otherwise, setting ORTHOG to .FALSE. will often lead to a reduction in solution time with negligible loss of reliability.

20: $X(N)$ -- DOUBLE PRECISION array
Input/Output
On entry: an estimate of the solution. In the absence of better information all elements of $X$ may be set to zero. On exit: from E04NAF, $X$ contains the best estimate of the solution.

21: $ISTATE(NCCTOTL)$ -- INTEGER array
Input/Output
On entry: with COLD as .FALSE., $ISTATE$ must indicate the status of every constraint with respect to the working set. The ordering of $ISTATE$ is as follows; the first $n$ elements of $ISTATE$ refer to the upper and lower bounds on the variables and elements $n+1$ through $n + NCLIN$ refer to the upper and lower bounds on $Ax$. The significance of each possible value of $ISTATE(j)$ is as follows:

<table>
<thead>
<tr>
<th>$ISTATE(j)$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>The constraint violates its lower bound by more than FEATOL($j$). This value of $ISTATE$ cannot occur after a feasible point has been found.</td>
</tr>
<tr>
<td>-1</td>
<td>The constraint violates its upper bound by more than FEATOL($j$). This value of $ISTATE$ cannot occur after a feasible point has been found.</td>
</tr>
<tr>
<td>0</td>
<td>The constraint is not in the working set. Usually, this means that the constraint lies strictly between its bounds.</td>
</tr>
<tr>
<td>1</td>
<td>This inequality constraint is included in the working set at its lower bound. The value of the constraint is within FEATOL($j$) of its lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>This inequality constraint is included in the working set at its upper bound. The value of the constraint is within FEATOL($j$) of its upper bound.</td>
</tr>
</tbody>
</table>
3 The constraint is included in the working set as an equality. This value of ISTATE can occur only when BL(j) = BU(j). The corresponding constraint is within FEATOL(j) of its required value.

If COLD = .TRUE., ISTATE need not be set by the user. However, when COLD = .FALSE., every element of ISTATE must be set to one of the values given above to define a suggested initial working set (which will be changed by E04NAF if necessary). The most likely values are:

<table>
<thead>
<tr>
<th>ISTATE(j)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The corresponding constraint should not be in the initial working set.</td>
</tr>
<tr>
<td>1</td>
<td>The constraint should be in the initial working set at its lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>The constraint should be in the initial working set at its upper bound.</td>
</tr>
<tr>
<td>3</td>
<td>The constraint should be in the initial working set as an equality. This value must not be specified unless BL(j) = BU(j). The values 1, 2 or 3 all have the same effect when BL(j) = BU(j).</td>
</tr>
</tbody>
</table>

Note that if E04NAF has been called previously with the same values of N and NCLIN, ISTATE already contains satisfactory values. On exit: when E04NAF exits with IFAIL set to 0, 1 or 3, the values in the array ISTATE indicate the status of the constraints in the active set at the solution. Otherwise, ISTATE indicates the composition of the working set at the final iterate.

22: ITER -- INTEGER Output
On exit: the number of iterations performed in either the LP phase or the QP phase, whichever was last entered.

Note that ITER is reset to zero after the LP phase.

23: OBJ -- DOUBLE PRECISION Output
On exit: the value of the quadratic objective function at x if x is feasible (IFAIL <= 5), or the sum of infeasibilities at x otherwise (6 <= IFAIL <= 8).

24: CLAMDA(NCTOTL) -- DOUBLE PRECISION array Output
On exit: the values of the Lagrange multiplier for each constraint with respect to the current working set. The ordering of CLAMDA is as follows; the first n components contain the multipliers for the bound constraints on the variables, and the remaining components contain the
multipliers for the general linear constraints. If ISTATE(j) = 0 (i.e., constraint j is not in the working set), CLAMDA(j) is zero. If x is optimal and ISTATE(j) = 1, CLAMDA(j) should be non-negative; if ISTATE(j) = 2, CLAMDA(j) should be non-positive.

25: IWORK(LIWORK) -- INTEGER array Workspace

26: LIWORK -- INTEGER Input
On entry:
the dimension of the array IWORK as declared in the (sub)program from which E04NAF is called.
Constraint: LIWORK>=2*N.

27: WORK(LWORK) -- DOUBLE PRECISION array Workspace

28: LWORK -- INTEGER Input
On entry:
the dimension of the array WORK as declared in the (sub)program from which E04NAF is called.
Constraints if LP = .FALSE. or NCLIN >= N then
LWORK>=2*N +4*N*NCLIN+NROWA.

if LP = .TRUE. and NCLIN < N then
LWORK>=2*(NCLIN+1) +4*N+2*NCLIN+NROWA.

If MSGLVL > 0, the amount of workspace provided and the amount of workspace required are output on the current advisory message unit (as defined by X04ABF). As an alternative to computing LWORK from the formula given above, the user may prefer to obtain an appropriate value from the output of a preliminary run with a positive value of MSGLVL and LWORK set to 1 (E04NAF will then terminate with IFAIL = 9).

29: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.
IFAIL contains zero on exit if \( x \) is a strong local minimum. i.e., the projected gradient is negligible, the Lagrange multipliers are optimal, and the projected Hessian is positive-definite. In some cases, a zero value of IFAIL means that \( x \) is a global minimum (e.g. when the Hessian matrix is positive-definite).

5.1. Description of the Printed Output

When MSGVL >= 5, a line of output is produced for every change in the working set (thus, several lines may be printed during a single iteration).

To aid interpretation of the printed results, we mention the convention for numbering the constraints: indices 1 through to \( n \) refer to the bounds on the variables, and when NCLIN > 0 indices \( n+1 \) through to \( n + \text{NCLIN} \) refer to the general constraints. When the status of a constraint changes, the index of the constraint is printed, along with the designation \( L \) (lower bound), \( U \) (upper bound) or \( E \) (equality).

In the LP phase, the printout includes the following:

- **ITN** is the iteration count.
- **JDEL** is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was deleted.
- **JADD** is the index of the constraint added to the working set. If JADD is zero, no constraint was added.
- **STEP** is the step taken along the computed search direction.
- **COND \( T \)** is a lower bound on the condition number of the matrix of predicted active constraints.
- **NUMINF** is the number of violated constraints (infeasibilities).
- **SUMINF** is a weighted sum of the magnitudes of the constraint violations.

\( T \)\n
- **LPOBJ** is the value of the linear objective function \( c \times \).
  It is printed only if LP = .TRUE..
During the QP phase, the printout includes the following:

**ITN** is the iteration count (reset to zero after the LP phase).

**JDEL** is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was deleted.

**JADD** is the index of the constraint added to the working set. If JADD is zero, no constraint was added.

**STEP** is the step (alpha) taken along the direction of $k$ search (if STEP is 1.0, the current point is a minimum in the subspace defined by the current working set).

**NHESS** is the number of calls to subroutine QPHESS.

**OBJECTIVE** is the value of the quadratic objective function.

**NCOLZ** is the number of columns of Z (see Section 3). In general, it is the dimension of the subspace in which the quadratic is currently being minimized.

**NORM GFREE** is the Euclidean norm of the gradient of the objective function with respect to the free variables, i.e. variables not currently held at a bound (NORM GFREE is not printed if ORTHOG = FALSE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

**NORM QTG** is a weighted norm of the gradient of the objective function with respect to the free variables (NORM QTG is not printed if ORTHOG = TRUE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

**NORM ZTG** is the Euclidean norm of the projected gradient (see Section 3).

**COND T** is a lower bound on the condition number of the
matrix of constraints in the working set.

**COND ZHZ** is a lower bound on the condition number of the projected Hessian matrix.

**HESS MOD** is the correction added to the diagonal of the projected Hessian to ensure that a satisfactory Cholesky factorization exists (see Section 3). When the projected Hessian is sufficiently positive-definite, HESS MOD will be zero.

When MSGLVL = 1 or MSGLVL >= 10, the summary printout at the end of execution of E04NAF includes a listing of the status of every constraint. Note that default names are assigned to all variables and constraints.

The following describes the printout for each variable.

**VARBL** is the name (V) and index j, j=1,2,...,n, of the variable.

**STATE** gives the state of the variable (FR if neither bound is in the working set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound, TB if held on a temporary bound). If VALUE lies outside the upper or lower bounds by more than FEATOL(j), STATE will be ++ or -- respectively.

**VALUE** is the value of the variable at the final iteration.

**LOWER BOUND** is the lower bound specified for the variable.

**UPPER BOUND** is the upper bound specified for the variable.

**LAGR MULT** is the value of the Lagrange multiplier for the associated bound constraint. This will be zero if STATE is FR. If x is optimal and STATE is LL, the multiplier should be non-negative; if STATE is UL, the multiplier should be non-positive.

**RESIDUAL** is the difference between the variable and the nearer of its bounds BL(j) and BU(j).

For each of the general constraints the printout is as above with refers to the jth element of Ax, except that VARBL is replaced by

**LNCON** The name (L) and index j, j=1,2,...,NCLIN, of the constraint.
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
x is a weak local minimum (the projected gradient is negligible, the Lagrange multipliers are optimal, but the projected Hessian is only semi-definite). This means that the solution is not unique.

IFAIL= 2
The solution appears to be unbounded, i.e., the quadratic function is unbounded below in the feasible region. This value of IFAIL occurs when a step of infinity would have to be taken in order to continue the algorithm.

IFAIL= 3
x appears to be a local minimum, but optimality cannot be verified because some of the Lagrange multipliers are very small in magnitude.

E04NAF has probably found a solution. However, the presence of very small Lagrange multipliers means that the predicted active set may be incorrect, or that x may be only a constrained stationary point rather than a local minimum. The method in E04NAF is not guaranteed to find the correct active set when there are very small multipliers. E04NAF attempts to delete constraints with zero multipliers, but this does not necessarily resolve the issue. The determination of the correct active set is a combinatorial problem that may require an extremely large amount of time. The occurrence of small multipliers often (but not always) indicates that there are redundant constraints.

IFAIL= 4
The iterates of the QP phase could be cycling, since a total of 50 changes were made to the working set without altering x.

This value will occur if 50 iterations are performed in the QP phase without changing x. The user should check the printed output for a repeated pattern of constraint deletions and additions. If a sequence of constraint changes is being repeated, the iterates are probably cycling. (E04NAF does not contain a method that is guaranteed to avoid cycling, which would be combinatorial in nature.) Cycling may occur in two circumstances: at a constrained stationary point where there are some small or zero Lagrange multipliers (see the discussion of IFAIL = 3); or at a point
(usually a vertex) where the constraints that are satisfied exactly are nearly linearly dependent. In the latter case, the user has the option of identifying the offending dependent constraints and removing them from the problem, or restarting the run with larger values of FEATOL for nearly dependent constraints. If E04NAF terminates with IFAIL = 4, but no suspicious pattern of constraint changes can be observed, it may be worthwhile to restart with the final x (with or without the warm start option).

IFAIL = 5
The limit of ITMAX iterations was reached in the QP phase before normal termination occurred.

The value of ITMAX may be too small. If the method appears to be making progress (e.g. the objective function is being satisfactorily reduced), increase ITMAX and rerun E04NAF (possibly using the warm start facility to specify the initial working set). If ITMAX is already large, but some of the constraints could be nearly linearly dependent, check the output for a repeated pattern of constraints entering and leaving the working set. (Near-dependencies are often indicated by wide variations in size in the diagonal elements of the T matrix, which will be printed if MSGLEV >= 30.) In this case, the algorithm could be cycling (see the comments for IFAIL = 4).

IFAIL = 6
The LP phase terminated without finding a feasible point, and hence it is not possible to satisfy all the constraints to within the tolerances specified by the FEATOL array. In this case, the final iterate will reveal values for which there will be a feasible point (e.g. a feasible point will exist if the feasibility tolerance for each violated constraint exceeds its RESIDUAL at the final point). The modified problem (with altered values in FEATOL) may then be solved using a warm start.

The user should check that there are no constraint redundancies. If the data for the jth constraint are accurate only to the absolute precision (delta), the user should ensure that the value of FEATOL(j) is greater than (delta). For example, if all elements of A are of order unity and are accurate only to three decimal places, every -3 component of FEATOL should be at least 10 .

IFAIL = 7
The iterates may be cycling during the LP phase; see the comments above under IFAIL = 4.
IFAIL = 8
The limit of ITMAX iterations was reached during the LP phase. See comments above under IFAIL = 5.

IFAIL = 9
An input parameter is invalid.

Overflow
If the printed output before the overflow error contains a warning about serious ill-conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of FEATOL(j) and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint (with index j) must be removed from the problem. If a warning message did not precede the fatal overflow, the user should contact NAG.

7. Accuracy
The routine implements a numerically stable active set strategy and returns solutions that are as accurate as the condition of the QP problem warrants on the machine.

8. Further Comments
The number of iterations depends upon factors such as the number of variables and the distances of the starting point from the solution. The number of operations performed per iteration is roughly proportional to \( (NFREE)^2 \), where NFREE (NFREE <= n) is the number of variables fixed on their upper or lower bounds.

Sensible scaling of the problem is likely to reduce the number of iterations required and make the problem less sensitive to perturbations in the data, thus improving the condition of the QP problem. See the Chapter Introduction and Gill et al [1] for further information and advice.

9. Example

\[
\begin{bmatrix}
1 & 1 & T
\end{bmatrix}
\]

To minimize the function \( c^T x + x^T H x \), where

\[
2
\]

\[
c = [-0.02, -0.2, -0.2, -0.2, -0.2, 0.04, 0.04, 2 0 0 0 0 0 0]
\]
subject to the bounds

```plaintext
[0 2 0 0 0 0 0]
[0 0 2 2 0 0 0]
H=[0 0 2 2 0 0 0]
[0 0 0 0 2 0 0]
[0 0 0 0 0 -2 -2]
[0 0 0 0 0 -2 -2]
```

---

**1. Purpose**

E04UCF is designed to minimize an arbitrary smooth function subject to constraints, which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints. (E04UCF may be used for unconstrained, bound-constrained and linearly constrained optimization.) The user must provide subroutines that define the objective and constraint functions and as many of their first partial derivatives as possible. Unspecified derivatives are approximated by finite differences. All matrices are treated as dense, and hence E04UCF is not intended for large sparse problems.

E04UCF uses a sequential quadratic programming (SQP) algorithm in which the search direction is the solution of a quadratic programming (QP) problem. The algorithm treats bounds, linear constraints and nonlinear constraints separately.

**2. Specification**

```plaintext
SUBROUTINE E04UCF (N, NCLIN, NCNLN, NROWA, NROWJ, NROWR,
1 A, BL, BU, CONFUN, OBJFUN, ITER,
2 ISTATE, C, CJAC, CLAMDA, OBJF, OBJGRD,
3 R, X, IWORK, LIWORK, WORK, LWORK,
```
3. Description

E04UCF is designed to solve the nonlinear programming problem -- the minimization of a smooth nonlinear function subject to a set of constraints on the variables. The problem is assumed to be stated in the following form:

\[
\begin{align*}
\text{Minimize} & \quad F(x) \quad \text{subject to} \quad l \leq (Ax) \leq u, \\
& \quad x \in \mathbb{R} \{c(x)\}
\end{align*}
\]

where \( F(x) \), the objective function, is a nonlinear function, \( A \) is an \( n \) by \( n \) constant matrix, and \( c(x) \) is an \( n \) element vector of nonlinear constraint functions. (The matrix \( A \) and the vector \( c(x) \) may be empty.) The objective function and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (The method of E04UCF will usually solve (1) if there are only isolated discontinuities away from the solution.)

This routine is essentially identical to the subroutine SOL/NPSOL described in Gill et al [8].

Note that upper and lower bounds are specified for all the variables and for all the constraints.

An equality constraint can be specified by setting \( l = u \). If
certain bounds are not present, the associated elements of l or u

\[ \begin{align*}
L & \text{ can be set to special values that will be treated as -infty or +infty.}
\end{align*} \]

If there are no nonlinear constraints in (1) and F is linear or

\[ \begin{align*}
\text{quadratic then one of E04MBF, E04NAF or E04NCF(*) will generally}
\end{align*} \]

be more efficient. If the problem is large and sparse the MINOS

\[ \begin{align*}
\text{package (see Murtagh and Saunders [13]) should be used, since}
\end{align*} \]

E04UCF treats all matrices as dense.

The user must supply an initial estimate of the solution to (1),

\[ \begin{align*}
\text{together with subroutines that define F(x), c(x) and as many}
\end{align*} \]

first partial derivatives as possible; unspecified derivatives

\[ \begin{align*}
\text{are approximated by finite differences.}
\end{align*} \]

The objective function is defined by subroutine OBJFUN, and the

\[ \begin{align*}
\text{nonlinear constraints are defined by subroutine CONFUN. On every}
\end{align*} \]

call, these subroutines must return appropriate values of the

\[ \begin{align*}
\text{objective and nonlinear constraints. The user should also provide}
\end{align*} \]

the available partial derivatives. Any unspecified derivatives

\[ \begin{align*}
\text{are approximated by finite differences; see Section 5.1 for a}
\end{align*} \]

discussion of the optional parameter Derivative Level. Just

\[ \begin{align*}
\text{before either OBJFUN or CONFUN is called, each element of the}
\end{align*} \]

current gradient array OBJGRD or CJAC is initialised to a special

\[ \begin{align*}
\text{value. On exit, any element that retains the value is estimated}
\end{align*} \]

by finite differences. Note that if there are nonlinear

\[ \begin{align*}
\text{costraints, then the first call to CONFUN will precede the first}
\end{align*} \]

call to OBJFUN.

For maximum reliability, it is preferable for the user to provide

\[ \begin{align*}
\text{all partial derivatives (see Chapter 8 of Gill et al [10], for a}
\end{align*} \]

detailed discussion). If all gradients cannot be provided, it is

\[ \begin{align*}
\text{similarly advisable to provide as many as possible. While}
\end{align*} \]

developing the subroutines OBJFUN and CONFUN, the optional

\[ \begin{align*}
\text{parameter Verify (see Section 5.1) should be used to check the}
\end{align*} \]

calculation of any known gradients.

E04UCF implements a sequential quadratic programming (SQP)

\[ \begin{align*}
\text{method. The document for E04NCF(*) should be consulted in}
\end{align*} \]

conjunction with this document.

In the rest of this section we briefly summarize the main

\[ \begin{align*}
\text{features of the method of E04UCF. Where possible, explicit}
\end{align*} \]

\[ \begin{align*}
\text{reference is made to the names of variables that are parameters}
\end{align*} \]

\[ \begin{align*}
\text{of subroutines E04UCF or appear in the printed output (see}
\end{align*} \]

Section 5.2).

At a solution of (1), some of the constraints will be active,

\[ \begin{align*}
\text{i.e., satisfied exactly. An active simple bound constraint}
\end{align*} \]
implies that the corresponding variable is fixed at its bound, and hence the variables are partitioned into fixed and free variables. Let $C$ denote the $m$ by $n$ matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by $n_{FX}$, with $n_{FR} = n - n_{FX}$ the number of free variables. The subscripts 'FX' and 'FR' on a vector or matrix will denote the vector or matrix composed of the components corresponding to fixed or free variables.

A point $x$ is a first-order Kuhn-Tucker point for (1) (see, e.g., Powell [14]) if the following conditions hold:

(i) $x$ is feasible;

(ii) there exist vectors $(x_i)$ and $(\lambda)$ (the Lagrange multiplier vectors for the bound and general constraints) such that

$$
T \quad g = C (\lambda) + (x_i), 
$$

where $g$ is the gradient of $F$ evaluated at $x$, and $(x_i) = 0$ if the $j$th variable is free.

(iii) The Lagrange multiplier corresponding to an inequality constraint active at its lower bound must be non-negative, and non-positive for an inequality constraint active at its upper bound.

Let $Z$ denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of $C$; i.e., $C Z = 0$. An equivalent statement of the condition (2) in terms of $Z$ is

$$
T \quad Z g_{FR} = 0.
$$

The vector $Z g_{FR}$ is termed the projected gradient of $F$ at $x$.

Certain additional conditions must be satisfied in order for a first-order Kuhn-Tucker point to be a solution of (1) (see, e.g., Powell [14]).

The method of E04UCF is a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher [5], Gill et al [10] and Powell [15].

The basic structure of E04UCF involves major and minor
iterations. The major iterations generate a sequence of iterates 
\( \{x_k\} \) that converge to \( x_k \), a first-order Kuhn-Tucker point of (1).

At a typical major iteration, the new iterate \( x \) is defined by

\[
x = x + (\alpha)p
\]

where \( x \) is the current iterate, the non-negative scalar \( \alpha \) is the step length, and \( p \) is the search direction. (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Also associated with each major iteration are estimates of the Lagrange multipliers and a prediction of the active set.

The search direction \( p \) in (3) is the solution of a quadratic programming subproblem of the form

\[
\begin{align*}
\text{Minimize} & \quad g^T p + \frac{1}{2} p^T H p, \\
\text{subject to} & \quad l \leq \{A p\} \leq u,
\end{align*}
\]

where \( g \) is the gradient of \( F \) at \( x \), the matrix \( H \) is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 8.3), and \( A \) is the Jacobian matrix of \( c \) evaluated at \( x \). (Finite-difference estimates may be used for \( g \) and \( A \); see the optional parameter Derivative Level in Section 5.1.) Let \( l \) in (1) be partitioned into three sections: \( l_B, l_L \) and \( l_N \), corresponding to the bound, linear and nonlinear constraints. The vector \( l \) in (4) is similarly partitioned, and is defined as

\[
\begin{align*}
l & = l - x, \\
l_B & = l_B - A x, \quad \text{and} \quad l_N = l_N - c,
\end{align*}
\]

where \( c \) is the vector of nonlinear constraints evaluated at \( x \).

The vector \( u \) is defined in an analogous fashion.

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly
for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities Bnd, Lin and Nln in the printed output of E04UCF.) In E04UCF, (4) is solved using E04NCF(*). Since solving a quadratic program as an iterative procedure, the minor iterations of E04UCF are the iterations of E04NCF(*). (More details about solving the subproblem are given in Section 8.1.)

Certain matrices associated with the QP subproblem are relevant in the major iterations. Let the subscripts 'FX' and 'FR' refer to the predicted fixed and free variables, and let C denote the m by n matrix of gradients of the general linear and nonlinear constraints in the predicted active set. First, we have available the TQ factorization of C:

$$C Q = (0 \ T),$$

$$C \quad FR$$

where T is a nonsingular m by m reverse-triangular matrix (i.e., t = 0 if i+j<m), and the non-singular n by n matrix Q is the product of orthogonal transformations (see Gill et al [6]).

Second, we have the upper-triangular Cholesky factor R of the transformed and re-ordered Hessian matrix

$$R = H^{\frac{1}{2}} Q Q^{\frac{1}{2}},$$

where H is the Hessian H with rows and columns permuted so that the free variables are first, and Q is the n by n matrix

$$Q = (I \ 0)$$

with I the identity matrix of order n. If the columns of Q are partitioned so that

$$Q = (Z \ Y),$$

the n (n=n-m) columns of Z form a basis for the null space of C. The matrix Z is used to compute the projected gradient Z T C.
at the current iterate. (The values Nz, Norm Gf and Norm Gz
printed by E04UCF give n and the norms of g and Z g .)

A theoretical characteristic of SQP methods is that the predicted
active set from the QP subproblem (4) is identical to the correct
active set in a neighbourhood of x . In E04UCF, this feature is
exploited by using the QP active set from the previous iteration
as a prediction of the active set for the next QP subproblem,
which leads in practice to optimality of the subproblems in only
one iteration as the solution is approached. Separate treatment
of bound and linear constraints in E04UCF also saves computation
in factorizing C and H .

Once p has been computed, the major iteration proceeds by
determining a step length (alpha) that produces a 'sufficient
decrease' in an augmented Lagrangian merit function (see Section
8.2). Finally, the approximation to the transformed Hessian
matrix H is updated using a modified BFGS quasi-Newton update
(see Section 8.3) to incorporate new curvature information
obtained in the move from x to x.

On entry to E04UCF, an iterative procedure from E04NCF(*) is
executed, starting with the user-provided initial point, to find
a point that is feasible with respect to the bounds and linear
constraints (using the tolerance specified by Linear Feasibility
Tolerance see Section 5.1). If no feasible point exists for the
bound and linear constraints, (1) has no solution and E04UCF
terminates. Otherwise, the problem functions will thereafter be
evaluated only at points that are feasible with respect to the
bounds and linear constraints. The only exception involves
variables whose bounds differ by an amount comparable to the
finite-difference interval (see the discussion of Difference
Interval in Section 5.1). In contrast to the bounds and linear
constraints, it must be emphasised that the nonlinear constraints
will not generally be satisfied until an optimal point is
reached.

Facilities are provided to check whether the user-provided
gradients appear to be correct (see the optional parameter Verify
in Section 5.1). In general, the check is provided at the first
point that is feasible with respect to the linear constraints and
bounds. However, the user may request that the check be performed
at the initial point.
In summary, the method of E04UCF first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem;

(b) a linesearch with an augmented Lagrangian merit function; and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian function.

These three procedures are described in more detail in Section 8.

4. References


5. Parameters

1: N -- INTEGER Input
   On entry: the number, n, of variables in the problem.
   Constraint: N > 0.

2: NCLIN -- INTEGER Input
   On entry: the number, n, of general linear constraints in the problem. Constraint: NCLIN >= 0.

3: NCNLN -- INTEGER Input
   On entry: the number, n, of nonlinear constraints in the problem. Constraint: NCNLN >= 0.

4: NROWA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which E04UCF is called.
CHAPTER 15. CHAPTER N

Constraint: \( \text{NROWA} \geq \text{max}(1, \text{NCLIN}) \).

5: \( \text{NROWJ} \) -- INTEGER Input
On entry:
the first dimension of the array CJAC as declared in the
(sub)program from which E04UCF is called.
Constraint: \( \text{NROWJ} \geq \text{max}(1, \text{NCNLN}) \).

6: \( \text{NROWR} \) -- INTEGER Input
On entry:
the first dimension of the array R as declared in the
(sub)program from which E04UCF is called.
Constraint: \( \text{NROWR} \geq N \).

7: \( A(\text{NROWA,*}) \) -- DOUBLE PRECISION array Input
The second dimension of the array A must be \( \geq N \) for \( \text{NCLIN} > 0 \). On entry: the ith row of the array A must contain the ith
row of the matrix A of general linear constraints in (1).
That is, the ith row contains the coefficients of the ith
general linear constraint, for \( i = 1, 2, \ldots, \text{NCLIN} \).
If \( \text{NCLIN} = 0 \) then the array A is not referenced.

8: \( \text{BL}(N+NCLIN+NCNLN) \) -- DOUBLE PRECISION array Input
On entry: the lower bounds for all the constraints, in the
following order. The first n elements of BL must contain the
lower bounds on the variables. If \( \text{NCLIN} > 0 \), the next n
elements of BL must contain the lower bounds on the general
linear constraints. If \( \text{NCNLN} > 0 \), the next n elements of BL
must contain the lower bounds for the nonlinear constraints.
To specify a non-existent lower bound (i.e., \( l = \infty \)), the
value used must satisfy \( BL(j) \leq -\text{BIGBND} \), where BIGBND is the
value of the optional parameter Infinite Bound Size whose
default value is 10 (see Section 5.1). To specify the jth
constraint as an equality, the user must set \( BL(j) = BU(j) = (\beta) \), say, where \( |(\beta)| < \text{BIGBND} \). Constraint: \( BL(j) \leq BU(j) \), for \( j=1, 2, \ldots, N+N\text{CLIN}+\text{NCNLN} \).

9: \( \text{BU}(N+NCLIN+NCNLN) \) -- DOUBLE PRECISION array Input
On entry: the upper bounds for all the constraints in the
following order. The first n elements of BU must contain the
upper bounds on the variables. If \( \text{NCLIN} > 0 \), the next n
elements of BU must contain the upper bounds on the general
linear constraints. If \( \text{NCNLN} > 0 \), the next n elements of BU
N
must contain the upper bounds for the nonlinear constraints. To specify a non-existent upper bound (i.e., \( u = +\infty \)), the value used must satisfy \( B_u(j) \geq B_l(j) \geq \text{BIGBND}, \) where \text{BIGBND} is the value of the optional parameter Infinite Bound Size, whose default value is 10 (see Section 5.1). To specify the \( j \)th constraint as an equality, the user must set \( B_u(j) = B_l(j) = (\beta), \) say, where \(|(\beta)| < \text{BIGBND}. \) Constraint: \( B_u(j) \geq B_l(j), \) for \( j=1,2,\ldots,N+NCLIN+NCNLN. \)

10: CONFUN -- SUBROUTINE, supplied by the user.

External Procedure

CONFUN must calculate the vector \( c(x) \) of nonlinear constraint functions and (optionally) its Jacobian for a specified \( n \) element vector \( x. \) If there are no nonlinear constraints (\( NCNLN=0 \)), CONFUN will never be called by E04UCF and CONFUN may be the dummy routine E04UDM. (E04UDM is included in the NAG Foundation Library and so need not be supplied by the user. Its name may be implementation-dependent: see the Users’ Note for your implementation for details.) If there are nonlinear constraints, the first call to CONFUN will occur before the first call to OBJFUN.

Its specification is:

```fortran
SUBROUTINE CONFUN (MODE, NCNLN, N, NROWJ, NEEDC,
1     X, C, CJAC, NSTATE, IUSER,
2     USER)

INTEGER MODE, NCNLN, N, NROWJ, NEEDC
1   (NCNLN), NSTATE, IUSER(*)
   DOUBLE PRECISION X(N), C(NCNLN), CJAC(NROWJ,N),
1   USER(*)

1: MODE -- INTEGER

Input/Output

On entry: MODE indicates the values that must be assigned during each call of CONFUN. MODE will always have the value 2 if all elements of the Jacobian are available, i.e., if Derivative Level is either 2 or 3 (see Section 5.1). If some elements of CJAC are unspecified, E04UCF will call CONFUN with MODE = 0, 1, or 2:

If MODE = 2, only the elements of C corresponding to positive values of NEEDC must be set (and similarly for the available components of the rows of CJAC).

If MODE = 1, the available components of the rows of CJAC corresponding to positive values in NEEDC must be
set. Other rows of CJAC and the array C will be ignored.

If MODE = 0, the components of C corresponding to positive values in NEEDC must be set. Other components and the array CJAC are ignored. On exit: MODE may be set to a negative value if the user wishes to terminate the solution to the current problem. If MODE is negative on exit from CONFUN then E04UCF will terminate with IFAIL set to MODE.

2: NCNLN -- INTEGER Input
   On entry: the number, n, of nonlinear constraints.

3: N -- INTEGER Input
   On entry: the number, n, of variables.

4: NROWJ -- INTEGER Input
   On entry: the first dimension of the array CJAC.

5: NEEDC(NCNLN) -- INTEGER array Input
   On entry: the indices of the elements of C or CJAC that must be evaluated by CONFUN. If NEEDC(i)>0 then the ith element of C and/or the ith row of CJAC (see parameter MODE above) must be evaluated at x.

6: X(N) -- DOUBLE PRECISION array Input
   On entry: the vector x of variables at which the constraint functions are to be evaluated.

7: C(NCNLN) -- DOUBLE PRECISION array Output
   On exit: if NEEDC(i)>0 and MODE = 0 or 2, C(i) must contain the value of the ith constraint at x. The remaining components of C, corresponding to the non-positive elements of NEEDC, are ignored.

8: CJAC(NROWJ,N) -- DOUBLE PRECISION array Output
   On exit: if NEEDC(i)>0 and MODE = 1 or 2, the ith row of CJAC must contain the available components of the vector \((\nabla)C_i\) given by
   \[
   (\nabla)C_i = \left( \begin{array}{ccc} 
   \frac{\partial C_i}{\partial x_1} & \cdots & \frac{\partial C_i}{\partial x_n} 
   \end{array} \right)
   \]
   where \(\cdots\) is the partial derivative of the ith
ddx
j
constraint with respect to the jth variable, evaluated
at the point x. See also the parameter NSTATE below.
The remaining rows of CJAC, corresponding to non-
positive elements of NEEDC, are ignored.

If all constraint gradients (Jacobian elements) are
known (i.e., Derivative Level = 2 or 3; see Section 5.1)
y any constant elements may be assigned to CJAC one
time only at the start of the optimization. An element
of CJAC that is not subsequently assigned in CONFUN
will retain its initial value throughout. Constant
elements may be loaded into CJAC either before the call
to E04UCF or during the first call to CONFUN (signalled
by the value NSTATE = 1). The ability to preload
constants is useful when many Jacobian elements are
identically zero, in which case CJAC may be initialised
to zero and non-zero elements may be reset by CONFUN.

Note that constant non-zero elements do affect the
values of the constraints. Thus, if CJAC(i,j) is set to
a constant value, it need not be reset in subsequent
calls to CONFUN, but the value CJAC(i,j)*X(j) must
nonetheless be added to C(i).

It must be emphasized that, if Derivative Level < 2,
unassigned elements of CJAC are not treated as
constant; they are estimated by finite differences, at
non-trivial expense. If the user does not supply a
value for Difference Interval (see Section 5.1), an
interval for each component of x is computed
automatically at the start of the optimization. The
automatic procedure can usually identify constant
elements of CJAC, which are then computed once only by
finite differences.

9: NSTATE -- INTEGER Input
On entry: if NSTATE = 1 then E04UCF is calling CONFUN
for the first time. This parameter setting allows the
user to save computation time if certain data must be
read or calculated only once.

10: IUSER(*) -- INTEGER array User Workspace

11: USER(*) -- DOUBLE PRECISION array User Workspace
CONFUN is called from E04UCF with the parameters IUSER
and USER as supplied to E04UCF. The user is free to use
the arrays IUSER and USER to supply information to
CONFUN as an alternative to using COMMON.
CONFUN must be declared as EXTERNAL in the (sub)program from which E04UCF is called. Parameters denoted as Input must not be changed by this procedure.

11: OBJFUN -- SUBROUTINE, supplied by the user.

**External Procedure**

OBJFUN must calculate the objective function \( F(x) \) and (optionally) the gradient \( g(x) \) for a specified \( n \) element vector \( x \).

Its specification is:

```fortran
SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD, 
                   NSTATE, IUSER, USER)
  INTEGER MODE, N, NSTATE, IUSER(*)
  DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)
```

1: **MODE** -- INTEGER  
Input/Output

On entry: MODE indicates the values that must be assigned during each call of OBJFUN.

MODE will always have the value 2 if all components of the objective gradient are specified by the user, i.e., if Derivative Level is either 1 or 3. If some gradient elements are unspecified, E04UCF will call OBJFUN with MODE = 0, 1 or 2.

- If MODE = 2, compute OBJF and the available components of OBJGRD.
- If MODE = 1, compute all available components of OBJGRD; OBJF is not required.
- If MODE = 0, only OBJF needs to be computed; OBJGRD is ignored.

On exit: MODE may be set to a negative value if the user wishes to terminate the solution to the current problem. If MODE is negative on exit from OBJFUN, then E04UCF will terminate with IFAIL set to MODE.

2: **N** -- INTEGER  
Input

On entry: the number, \( n \), of variables.

3: **X(N)** -- DOUBLE PRECISION array  
Input

On entry: the vector \( x \) of variables at which the objective function is to be evaluated.

4: **OBJF** -- DOUBLE PRECISION  
Output

On exit: if MODE = 0 or 2, OBJF must be set to the value of the objective function at \( x \).
5: OBJGRD(N) -- DOUBLE PRECISION array  Output
   On exit: if MODE = 1 or 2, OBJGRD must return the
   available components of the gradient evaluated at x.

6: NSTATE -- INTEGER  Input
   On entry: if NSTATE = 1 then E04UCF is calling OBJFUN
   for the first time. This parameter setting allows the
   user to save computation time if certain data must be
   read or calculated only once.

7: IUSER(*) -- INTEGER array  User Workspace

8: USER(*) -- DOUBLE PRECISION array  User Workspace
   OBJFUN is called from E04UCF with the parameters IUSER
   and USER as supplied to E04UCF. The user is free to use
   the arrays IUSER and USER to supply information to
   OBJFUN as an alternative to using COMMON.
   OBJFUN must be declared as EXTERNAL in the (sub)program
   from which E04UCF is called. Parameters denoted as
   Input must not be changed by this procedure.

12: ITER -- INTEGER  Output
   On exit: the number of iterations performed.

13: ISTATE(N+NCLIN+NCNLN) -- INTEGER array  Input/Output
   On entry: ISTATE need not be initialised if E04UCF is called
   with (the default) Cold Start option. The ordering of ISTATE
   is as follows. The first n elements of ISTATE refer to the
   upper and lower bounds on the variables, elements n+1
   through n+n refer to the upper and lower bounds on A x, and
   L
   elements n+n+1 through n+n+n refer to the upper and lower
   L N
   bounds on c(x). When a Warm Start option is chosen, the
   elements of ISTATE corresponding to the bounds and linear
   constraints define the initial working set for the procedure
   that finds a feasible point for the linear constraints and
   bounds. The active set at the conclusion of this procedure
   and the elements of ISTATE corresponding to nonlinear
   constraints then define the initial working set for the
   first QP subproblem. Possible values for ISTATE(j) are:

   ISTATE(j)  Meaning
   0          The corresponding constraint is not in the initial
               QP working set.
   1          This inequality constraint should be in the
               working set at its lower bound.
This inequality constraint should be in the working set at its upper bound.

This equality constraint should be in the initial working set. This value must not be specified unless $BL(j) = BU(j)$. The values 1, 2 or 3 all have the same effect when $BL(j) = BU(j)$.

Note that if E04UCF has been called previously with the same values of N, NCLIN and NCNLN, ISTATE already contains satisfactory values. If necessary, E04UCF will override the user’s specification of ISTATE so that a poor choice will not cause the algorithm to fail. On exit: with IFAIL = 0 or 1, the values in the array ISTATE correspond to the active set of the final QP subproblem, and are a prediction of the status of the constraints at the solution of the problem. Otherwise, ISTATE indicates the composition of the QP working set at the final iterate. The significance of each possible value of ISTATE(j) is as follows:

-2 This constraint violates its lower bound by more than the appropriate feasibility tolerance (see the optional parameters LinearFeasibility Tolerance and Nonlinear Feasibility Tolerance in Section 5.1). This value can occur only when no feasible point can be found for a QP subproblem.

-1 This constraint violates its upper bound by more than the appropriate feasibility tolerance (see the optional parameters LinearFeasibility Tolerance and Nonlinear Feasibility Tolerance in Section 5.1). This value can occur only when no feasible point can be found for a QP subproblem.

0 The constraint is satisfied to within the feasibility tolerance, but is not in the working set.

1 This inequality constraint is included in the QP working set at its upper bound.

2 This inequality constraint is included in the QP working set at its upper bound.

3 This constraint is included in the QP working set as an equality. This value of ISTATE can occur only when $BL(j) = BU(j)$.

14: C(*) -- DOUBLE PRECISION array Output

Note: the dimension of the array C must be at least $\max(1, NCNLN)$.

On exit: if $NCNLN > 0$, C(i) contains the value of the ith
nonlinear constraint function \( c \) at the final iterate, for \( i = 1, 2, \ldots, NCNLN \). If \( NCNLN = 0 \), then the array \( C \) is not referenced.

15: \( CJAC(NROWJ,*) \) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array \( CJAC \) must be at least \( N \) for \( NCNLN > 0 \) and 1 otherwise On entry: in general, \( CJAC \) need not be initialised before the call to E04UCF. However, if Derivative Level = 3, the user may optionally set the constant elements of \( CJAC \) (see parameter NSTATE in the description of CONFUN). Such constant elements need not be re-assigned on subsequent calls to CONFUN. If \( NCNLN = 0 \), then the array \( CJAC \) is not referenced. On exit: if \( NCNLN > 0 \), \( CJAC \) contains the Jacobian matrix of the nonlinear constraint functions at the final iterate, i.e., \( CJAC(i,j) \) contains the partial derivative of the \( i \)th constraint function with respect to the \( j \)th variable, for \( i = 1, 2, \ldots, NCNLN; j = 1, 2, \ldots, N \). (See the discussion of parameter \( CJAC \) under CONFUN.)

16: \( CLAMDA(N+NCLIN+NCNLN) \) -- DOUBLE PRECISION array Input/Output
On entry: \( CLAMDA \) need not be initialised if E04UCF is called with the (default) Cold Start option. With the Warm Start option, \( CLAMDA \) must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by the ISTATE array (as above). The ordering of \( CLAMDA \) is as follows; the first \( n \) elements contain the multipliers for the bound constraints on the variables, elements \( n+1 \) through \( n+n \) contain the multipliers \( L \) for the general linear constraints, and elements \( n+n+1 \) through \( n+n+n \) contain the multipliers \( L N \) for the nonlinear constraints. If the \( j \)th constraint is defined as 'inactive' by the initial value of the ISTATE array, \( CLAMDA(j) \) should be zero; if the \( j \)th constraint is an inequality active at its lower bound, \( CLAMDA(j) \) should be non-negative; if the \( j \)th constraint is an inequality active at its upper bound, \( CLAMDA(j) \) should be non-positive. On exit: the values of the QP multipliers from the last QP subproblem. \( CLAMDA(j) \) should be non-negative if ISTATE\((j) = 1 \) and non-positive if ISTATE\((j) = 2 \).

17: \( OBJF \) -- DOUBLE PRECISION Output
On exit: the value of the objective function, \( F(x) \), at the final iterate.

18: \( OBJGRD(N) \) -- DOUBLE PRECISION array Output
On exit: the gradient (or its finite-difference approximation) of the objective function at the final iterate.

19: $\mathbf{R}(\mathrm{NROWR}, \mathrm{N})$ -- DOUBLE PRECISION array Input/Output
On entry: $\mathbf{R}$ need not be initialised if E04UCF is called with a Cold Start option (the default), and will be taken as the identity. With a Warm Start $\mathbf{R}$ must contain the upper-triangular Cholesky factor $\mathbf{R}$ of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements not in the upper-triangular part of $\mathbf{R}$ are assumed to be zero and need not be assigned.
On exit: if Hessian = No, (the default; see Section 5.1), $\mathbf{R}^T$ contains the upper-triangular Cholesky factor $\mathbf{R}$ of $\mathbf{Q}^T \mathbf{H} \mathbf{Q}$, an estimate of the transformed and re-ordered Hessian of the Lagrangian at $\mathbf{x}$ (see (6) in Section 3). If Hessian = Yes, $\mathbf{R}$ contains the upper-triangular Cholesky factor $\mathbf{R}$ of $\mathbf{H}$, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

20: $\mathbf{X}(\mathrm{N})$ -- DOUBLE PRECISION array Input/Output
On entry: an initial estimate of the solution. On exit: the final estimate of the solution.

21: $\mathbf{IWORK}(\mathrm{LIWORK})$ -- INTEGER array Workspace

22: $\mathrm{LIWORK}$ -- INTEGER Input
On entry: the dimension of the array IWORK as declared in the (sub)program from which E04UCF is called.
Constraint: $\mathrm{LIWORK} \geq 3*\mathrm{N} + \mathrm{NCLIN} + 2*\mathrm{NCNLN}$.

23: $\mathbf{WORK}(\mathrm{LWORK})$ -- DOUBLE PRECISION array Workspace

24: $\mathrm{LWORK}$ -- INTEGER Input
On entry: the dimension of the array WORK as declared in the (sub)program from which E04UCF is called.
Constraints:
if $\mathrm{NCLIN} = \mathrm{NCNLN} = 0$ then
\begin{align*}
\mathrm{LWORK} & \geq 20*\mathrm{N} \\
\end{align*}
if $\mathrm{NCNLN} = 0$ and $\mathrm{NCLIN} > 0$ then
\begin{align*}
\mathrm{LWORK} & \geq 2*\mathrm{N} + 20*\mathrm{N} + 11*\mathrm{NCLIN} \\
\end{align*}
if $\mathrm{NCNLN} > 0$ and $\mathrm{NCLIN} \geq 0$ then
\begin{align*}
\mathrm{LWORK} & \geq 2*\mathrm{N} + \mathrm{N} + 11*\mathrm{NCLIN} + 20*\mathrm{N} + 20*\mathrm{N} + 21*\mathrm{NCNLN} \\
\end{align*}
If Major Print Level > 0, the required amounts of workspace are output on the current advisory message channel (see X04ABF). As an alternative to computing LIWORK and LWORK from the formulas given above, the user may prefer to obtain appropriate values from the output of a preliminary run with a positive value of Major Print Level and LIWORK and LWORK set to 1. (E04UCF will then terminate with IFAIL = 9.)

25: IUSER(*) -- INTEGER array 
User Workspace
Note: the dimension of the array IUSER must be at least 1.
IUSER is not used by E04UCF, but is passed directly to routines CONFUN and OBJFUN and may be used to pass information to those routines.

26: USER(*) -- DOUBLE PRECISION array 
User Workspace
Note: the dimension of the array USER must be at least 1.
USER is not used by E04UCF, but is passed directly to routines CONFUN and OBJFUN and may be used to pass information to those routines.

27: IFAIL -- INTEGER 
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

E04UCF returns with IFAIL = 0 if the iterates have converged to a point x that satisfies the first-order Kuhn-Tucker conditions to the accuracy requested by the optional parameter Optimality Tolerance (see Section 5.1), i.e., the projected gradient and active constraint residuals are negligible at x.

The user should check whether the following four conditions are satisfied:
(i) the final value of Norm Gz is significantly less than that at the starting point;
(ii) during the final major iterations, the values of Step and ItQP are both one;
(iii) the last few values of both Norm Gz and Norm C become
small at a fast linear rate;

(iv) Cond Hz is small.
If all these conditions hold, x is almost certainly a local
minimum of (1). (See Section 9 for a specific example.)

5.1. Optional Input Parameters
Several optional parameters in E04UCF define choices in the
behaviour of the routine. In order to reduce the number of formal
parameters of E04UCF these optional parameters have associated
default values (see Section 5.1.3) that are appropriate for most
problems. Therefore the user need only specify those optional
parameters whose values are to be different from their default
values.

The remainder of this section can be skipped by users who wish to
use the default values for all optional parameters. A complete
list of optional parameters and their default values is given in
Section 5.1.3

5.1.1. Specification of the optional parameters
Optional parameters may be specified by calling one, or both, of
E04UDF and E04UEF prior to a call to E04UCF.

E04UDF reads options from an external options file, with Begin
and End as the first and last lines respectively and each
intermediate line defining a single optional parameter. For
example,

Begin
Print Level = 1
End

The call

CALL E04UDF (IOPTNS, INFORM)

can then be used to read the file on unit IOPTNS. INFORM will be
zero on successful exit. E04UDF should be consulted for a full
description of this method of supplying optional parameters.

E04UEF can be called directly to supply options, one call being
necessary for each optional parameter. For example,

CALL E04UEF ('Print level = 1')

E04UEF should be consulted for a full description of this method
of supplying optional parameters.
All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user are unaltered by E04UCF (unless they define invalid values) and so remain in effect for subsequent calls to E04UCF, unless altered by the user.

5.1.2. Description of the optional parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and DOUBLE PRECISION values required with certain options. The number (epsilon) is a generic notation for machine precision (see X02AJF(*) ), and (epsilon) denotes the relative precision of the objective function (the optional parameter Function Precision see below).

Central Difference Interval r Default values are computed

If the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate, the value of r is used as the difference interval for every component of x. The use of finite-differences is discussed further below under the optional parameter Difference Interval.

Cold Start Default = Cold Start

Warm Start

(AXIOM parameter STA, warm start when .TRUE.)

This option controls the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter. With a Cold Start, the first working set is chosen by E04UCF based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (within Crash Tolerance; see below). With a Warm Start, the user must set the ISTATE array and define CLAMDA and R as discussed in Section 5. ISTATE values associated with bounds and linear constraints determine the initial working set of the procedure to find a feasible point with respect to the bounds and linear constraints.
ISTATE values associated with nonlinear constraints determine the initial working set of the first QP subproblem after such a feasible point has been found. E04UCF will override the user's specification of ISTATE if necessary, so that a poor choice of the working set will not cause a fatal error. A warm start will be advantageous if a good estimate of the initial working set is available - for example, when E04UCF is called repeatedly to solve related problems.

Crash Tolerance $r$ Default = 0.01

(AXIOM parameter CRA)

This value is used in conjunction with the optional parameter Cold Start (the default value). When making a cold start, the QP algorithm in E04UCF must select an initial working set. When $r\geq 0$, the initial working set will include (if possible) bounds or general inequality constraints that lie within $r$ of their bounds.

In particular, a constraint of the form $a_j^T x \geq l_j$ will be included in the initial working set if $|a_j^T x - l_j| \leq r(1 + |l_j|)$. If $r < 0$ or $r > 1$, the default value is used.

Defaults

This special keyword may be used to reset the default values following a call to E04UCF.

Derivative Level $i$ Default = 3

(AXIOM parameter DER)

This parameter indicates which derivatives are provided by the user in subroutines OBJFUN and CONFUN. The possible choices for $i$ are the following.

<table>
<thead>
<tr>
<th>$i$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>All objective and constraint gradients are provided by the user.</td>
</tr>
<tr>
<td>2</td>
<td>All of the Jacobian is provided, but some components of the objective gradient are not specified by the user.</td>
</tr>
<tr>
<td>1</td>
<td>All elements of the objective gradient are known, but some elements of the Jacobian matrix are not specified by the user.</td>
</tr>
</tbody>
</table>
0 Some elements of both the objective gradient and the Jacobian matrix are not specified by the user.

The value $i=3$ should be used whenever possible, since E04UCF is more reliable and will usually be more efficient when all derivatives are exact.

If $i=0$ or 2, E04UCF will estimate the unspecified components of the objective gradient, using finite differences. The computation of finite-difference approximations usually increases the total run-time, since a call to OBJFUN is required for each unspecified element. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill et al [10], for a discussion of limiting accuracy).

If $i=0$ or 1, E04UCF will approximate unspecified elements of the Jacobian. One call to CONFUN is needed for each variable for which partial derivatives are not available. For example, if the Jacobian has the form

\[
\begin{pmatrix}
* & * & * & * \\
* & ? & ? & * \\
* & * & ? & * \\
* & * & * & *
\end{pmatrix}
\]

where '*' indicates an element provided by the user and '?' indicates an unspecified element, E04UCF will call CONFUN twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to CONFUN.)

At times, central differences are used rather than forward differences, in which case twice as many calls to OBJFUN and CONFUN are needed. (The switch to central differences is not under the user's control.)

Difference Interval $r$ Default values are computed

(AXIOM parameter DIF)

This option defines an interval used to estimate gradients by finite differences in the following circumstances:

(a) For verifying the objective and/or constraint gradients (see the description of Verify, below).

(b) For estimating unspecified elements of the objective gradient of the Jacobian matrix.
In general, a derivative with respect to the $j$th variable is approximated using the interval $(\delta_j)$, where $\delta_j = r(1 + |x_j|)$

with $x$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to $O(r)$.

See Gill et al [10] for a discussion of the accuracy in finite-difference approximations.

If a difference interval is not specified by the user, a finite-difference interval will be computed automatically for each variable by a procedure that requires up to six calls of CONFUN and OBJFUN for each component. This option is recommended if the function is badly scaled or the user wishes to have E04UCF determine constant elements in the objective and constraint gradients (see the descriptions of CONFUN and OBJFUN in Section 5).

Feasibility Tolerance $r$ Default $= \sqrt{\epsilon}$

(AXIOM parameter FEA)

The scalar $r$ defines the maximum acceptable absolute violations in linear and nonlinear constraints at a 'feasible' point; i.e., a constraint is considered satisfied if its violation does not exceed $r$. If $r<\epsilon$ or $r\geq 1$, the default value is used. Using this keyword sets both optional parameters Linear Feasibility Tolerance and Nonlinear Feasibility Tolerance to $r$, if $\epsilon\leq r<1$. (Additional details are given below under the descriptions of these parameters.)

0.9

Function Precision $r$ Default $= \epsilon$

(AXIOM parameter FUN)

This parameter defines $\epsilon$, which is intended to be a measure of the accuracy with which the problem functions $f$ and $c$ can be computed. If $r<\epsilon$ or $r\geq 1$, the default value is used. The value of $\epsilon$ should reflect the relative precision of $1+|F(x)|$; i.e., $\epsilon$ acts as a relative precision when $|F|$ is large, and as an absolute precision when $|F|$ is small. For example, if $F(x)$ is typically of order 1000 and
the first six significant digits are known to be correct, an appropriate value for (epsilon) would be 1.0E-6. In contrast, if 

\[ F(x) \text{ is typically of order } 10^{-4} \] and the first six significant digits are known to be correct, an appropriate value for (epsilon) would be 1.0E-10. The choice of (epsilon) can be quite complicated for badly scaled problems; see Chapter 8 of Gill et al [10] for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of (epsilon) should be large enough so that E04UCF will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Hessian No Default = No

Hessian Yes

(No AXIOM parameter - fixed as Yes)

This option controls the contents of the upper-triangular matrix R (see Section 5). E04UCF works exclusively with the transformed and re-ordered Hessian H (6), and hence extra computation is required to form the Hessian itself. If Hessian = No, R contains the Cholesky factor of the transformed and re-ordered Hessian. If Hessian = Yes the Cholesky factor of the approximate Hessian itself is formed and stored in R. The user should select Hessian = Yes if a warm start will be used for the next call to E04UCF.

10

Infinite Bound Size r Default = 10

(AXIOM parameter INFB)

If r>0, r defines the 'infinite' bound BIGBND in the definition of the problem constraints. Any upper bound greater than or equal to BIGBND will be regarded as plus infinity (and similarly for a lower bound less than or equal to -BIGBND). If r<=0, the default value is used.

10

Infinite Step Size r Default = max(BIGBND,10)

(AXIOM parameter INFS)
If $r > 0$, $r$ specifies the magnitude of the change in variables that is treated as a step to an unbounded solution. If the change in $x$ during an iteration would exceed the value of Infinite Step Size, the objective function is considered to be unbounded below in the feasible region. If $r \leq 0$, the default value is used.

Iteration limit $\text{i Default} = \max(50, 3(n + n^2) + 10n )$

See Major Iteration Limit below.

Linear Feasibility Tolerance $r$ Default = $\sqrt{\text{epsilon}}$

(AXIOM parameter LINF)

Nonlinear Feasibility Tolerance $r$ Default = $\sqrt{\text{epsilon}}$ if $2$

(AXIOM parameter NONF)

Derivative Level $\geq 2$ and $\text{epsilon}$ otherwise

The scalars $r_1$ and $r_2$ define the maximum acceptable absolute violations in linear and nonlinear constraints at a 'feasible' point; i.e., a linear constraint is considered satisfied if its violation does not exceed $r_1$, and similarly for a nonlinear constraint and $r_2$. If $r_1 < \text{epsilon}$ or $r_2 = 1$, the default value is used, for $i = 1, 2$.

On entry to E04UCF, an iterative procedure is executed in order to find a point that satisfies the linear constraint and bounds on the variables to within the tolerance $r_1$. All subsequent iterates will satisfy the linear constraints to within the same tolerance (unless $r_1$ is comparable to the finite-difference interval).

For nonlinear constraints, the feasibility tolerance $r_2$ defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, the value of Nonlinear Feasibility
Tolerance acts as a partial termination criterion for the iterative sequence generated by E04UCF (see the discussion of Optimal Tolerance).

These tolerances should reflect the precision of the corresponding constraints. For example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify $r$ as $10^{-6}$.

Linesearch Tolerance $r$ Default = 0.9

(AXIOM parameter LINT)

The value $r$ ($0 \leq r < 1$) controls the accuracy with which the step $(\alpha)$ taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of $r$, the more accurate the linesearch). The default value $r=0.9$ requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If there are no nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations - for example, if the objective function is cheap to evaluate, or if a substantial number of gradients are unspecified.

List Default = List

Nolist

(AXIOM parameter LIST)

Normally each optional parameter specification is printed as it is supplied. Nolist may be used to suppress the printing and List may be used to restore printing.

Major Iteration Limit $i$ Default = $\max(50,3(n+n)+10n)$

Iteration Limit

Iters

Itns

(AXIOM parameter MAJI)
The value of $i$ specifies the maximum number of major iterations allowed before termination. Setting $i=0$ and Major Print Level $> 0$ means that the workspace needed will be computed and printed, but no iterations will be performed.

Major Print level $i$ Default $= 10$

Print Level

(AXIOM parameter MAJP)

The value of $i$ controls the amount of printout produced by the major iterations of E04UCF. (See also Minor Print level below.) The levels of printing are indicated below.

- $i$ Output
  - 0 No output.
  - 1 The final solution only.
  - 5 One line for each major iteration (no printout of the final solution).
  - $\geq 10$ The final solution and one line of output for each iteration.
  - $\geq 20$ At each major iteration, the objective function, the Euclidean norm of the nonlinear constraint violations, the values of the nonlinear constraints (the vector $c$), the values of the linear constraints (the vector $Ax$), and the current values of the variables (the vector $x$).
  - $\geq 30$ At each major iteration, the diagonal elements of the matrix $T$ associated with the TQ factorization (5) of the QP working set, and the diagonal elements of $R$, the triangular factor of the transformed and re-ordered Hessian (6).

Minor Iteration Limit $i$ Default $= \max(50,3(n+n+n))$

(AXIOM parameter MINI)

The value of $i$ specifies the maximum number of iterations for the optimality phase of each QP subproblem.

Minor Print Level $i$ Default $= 0$
The value of \( i \) controls the amount of printout produced by the minor iterations of E04UCF, i.e., the iterations of the quadratic programming algorithm. (See also Major Print Level, above.) The following levels of printing are available.

- \( i = 0 \): No output.
- \( i = 1 \): The final QP solution.
- \( i = 5 \): One line of output for each minor iteration (no printout of the final QP solution).
- \( i \geq 10 \): The final QP solution and one brief line of output for each minor iteration.
- \( i \geq 20 \): At each minor iteration, the current estimates of the QP multipliers, the current estimate of the QP search direction, the QP constraint values, and the status of each QP constraint.
- \( i \geq 30 \): At each minor iteration, the diagonal elements of the matrix \( T \) associated with the TQ factorization (5) of the QP working set, and the diagonal elements of the Cholesky factor \( R \) of the transformed Hessian (6).

Nonlinear Feasibility Tolerance \( r \) Default = \( \sqrt{\text{epsilon}} \)

See Linear Feasibility Tolerance, above.

\[ r = 0.8 \]

Optimality Tolerance \( r \) Default = \( \text{epsilon} \)

The parameter \( r \) \( (\epsilon \leq r < 1) \) specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, \( r \) indicates the number of correct figures desired in the objective function at the solution. For example, if \( r \) is 10 and E04UCF terminates successfully, the final value of \( F \) should have approximately six correct figures. If \( r \leq \epsilon \) or \( r \geq 1 \) the default value is used.
E04UCF will terminate successfully if the iterative sequence of $x$ values is judged to have converged and the final point satisfies the first-order Kuhn-Tucker conditions (see Section 3). The sequence of iterates is considered to have converged at $x$ if

$$
\text{(alpha)} \|p\| \leq \sqrt{r(1+\|x\|)}, 
$$

(8a)

where $p$ is the search direction and (alpha) the step length from (3). An iterate is considered to satisfy the first-order conditions for a minimum if

$$
T \frac{\|Zg\|}{\|g\|} \leq \sqrt{r(1+\max(1+|F(x)|,\|g\|))} 
$$

(8b)

and

$$
|\text{res}_j| \leq \text{ftol} \text{ for all } j, 
$$

(8c)

where $Zg$ is the projected gradient (see Section 3), $g$ is the gradient of $F(x)$ with respect to the free variables, $\text{res}_j$ is the violation of the $j$th active nonlinear constraint, and ftol is the Nonlinear Feasibility Tolerance.

Step Limit $r$ Default = 2.0

(AXIOM parameter STE)

If $r>0$, $r$ specifies the maximum change in variables at the first step of the linesearch. In some cases, such as $F(x)=ae$ or $bF(x)=ax$, even a moderate change in the components of $x$ can lead to floating-point overflow. The parameter $r$ is therefore used to encourage evaluation of the problem functions at meaningful points. Given any major iterate $x$, the first point $x$ at which $F$ and $c$ are evaluated during the linesearch is restricted so that

$$
\|x-x\| \leq r(1+\|x\|). 
$$

(2)

The linesearch may go on and evaluate $F$ and $c$ at points further
from $x$ if this will result in a lower value of the merit function. In this case, the character L is printed at the end of the optional line of printed output, (see Section 5.2). If L is printed for most of the iterations, $r$ should be set to a larger value.

Wherever possible, upper and lower bounds on $x$ should be used to prevent evaluation of nonlinear functions at wild values. The default value Step Limit = 2.0 should not affect progress on well-behaved functions, but values 0.1 or 0.01 may be helpful when rapidly varying functions are present. If a small value of Step Limit is selected, a good starting point may be required. An important application is to the class of nonlinear least-squares problems. If $r \leq 0$, the default value is used.

Start Objective Check At Variable $k$ Default = 1

(AXIOM parameter STAO)

Start Constraint Check At Variable $k$ Default = 1

(AXIOM parameter STAC)

Stop Objective Check At Variable $l$ Default = $n$

(AXIOM parameter STOO)

Stop Constraint Check At Variable $l$ Default = $n$

(AXIOM parameter STOC)

These keywords take effect only if Verify Level > 0 (see below). They may be used to control the verification of gradient elements computed by subroutines OBJFUN and CONFUN. For example, if the first 30 components of the objective gradient appeared to be correct in an earlier run, so that only component 31 remains questionable, it is reasonable to specify Start Objective Check At Variable 31. If the first 30 variables appear linearly in the objective, so that the corresponding gradient elements are constant, the above choice would also be appropriate.

Verify Level $i$ Default = 0

Verify No

Verify Level - 1

Verify Level 0

Verify Objective Gradients
Verify Level 1
Verify Constraint Gradients
Verify Level 2
Verify
Verify Yes
Verify Gradients
Verify Level 3

(AXIOM parameter VE)

These keywords refer to finite-difference checks on the gradient elements computed by the user-provided subroutines OBJFUN and CONFUN. (Unspecified gradient components are not checked.) It is possible to specify Verify Levels 0-3 in several ways, as indicated above. For example, the nonlinear objective gradient (if any) will be verified if either Verify Objective Gradients or Verify Level 1 is specified. Similarly, the objective and the constraint gradients will be verified if Verify Yes or Verify Level 3 or Verify is specified.

If $0 \leq i \leq 3$, gradients will be verified at the first point that satisfies the linear constraints and bounds. If $i=0$, only a 'cheap' test will be performed, requiring one call to OBJFUN and one call to CONFUN. If $1 \leq i \leq 3$, a more reliable (but more expensive) check will be made on individual gradient components, within the ranges specified by the Start and Stop keywords described above. A result of the form OK or BAD? is printed by E04UCF to indicate whether or not each component appears to be correct.

If $10 \leq i \leq 13$, the action is the same as for $i - 10$, except that it will take place at the user-specified initial value of $x$.

We suggest that Verify Level 3 be specified whenever a new function routine is being developed.

5.1.3. Optional parameter checklist and default values

For easy reference, the following list shows all the valid keywords and their default values. The symbol (epsilon) represents the machine precision (see X02AJF(*) ).

Optional Parameters    Default Values
Central difference interval

Cold/Warm start Cold start
Crash tolerance 0.01

Defaults

Derivative level 3
Difference interval Computed automatically

Feasibility tolerance \sqrt{\text{epsilon}}

0.9
Function precision \text{epsilon}
Hessian No

Infinite bound size 10
Infinite step size 10

Linear feasibility tolerance
Linesearch tolerance 0.9
List/Nolist List

Major iteration limit \max(50, 3(n+n) + 10n )

L N

Major print level 10

Minor iteration limit \max(50, 3(n+n + n ))

L N

Minor print level 0

Nonlinear feasibility \sqrt{\text{epsilon}} \text{ if Derivative Level } \geq 2
tolerance 0.33
otherwise (epsilon)
Optimality tolerance 0.8
Step limit 2.0
Start objective check 1
Start constraint check 1
Stop objective check n
Stop constraint check n
Verify level 0

5.2. Description of Printed Output

The level of printed output from E04UCF is controlled by the user (see the description of Major Print Level and Minor Print Level in Section 5.1). If Minor Print Level > 0, output is obtained from the subroutines that solve the QP subproblem. For a detailed description of this information the reader should refer to E04NCF(*)

When Major Print Level >= 5, the following line of output is produced at every major iteration of E04UCF. In all cases, the values of the quantities printed are those in effect on completion of the given iteration.

Itn is the iteration count.

ItQP is the sum of the iterations required by the feasibility and optimality phases of the QP subproblem. Generally, ItQP will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 3).

Note that ItQP may be greater than the Minor Iteration Limit if some iterations are required for the feasibility phase.

Step is the step (alpha) taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.
Nfun is the cumulative number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the gradients by finite differences are not included. Nfun is printed as a guide to the amount of work required for the linesearch.

Merit is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 8.2). As the solution is approached, Merit will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by I at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of Merit values will decrease monotonically until either a feasible subproblem is obtained or E04UCF terminates with IFAIL = 3 (no feasible point could be found for the nonlinear constraints).

If no nonlinear constraints are present (i.e., NCNLN = 0), this entry contains Objective, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

Bnd is the number of simple bound constraints in the predicted active set.

Lin is the number of general linear constraints in the predicted active set.

Nln is the number of nonlinear constraints in the predicted active set (not printed if NCNLN is zero).

Nz is the number of columns of Z (see Section 8.1). The value of Nz is the number of variables minus the number of constraints in the predicted active set; i.e., $Nz = n - (Bnd + Lin + Nln)$. 
Norm $G_f$ is the Euclidean norm of $g$, the gradient of the objective function with respect to the free variables, i.e., variables not currently held at a bound.

$T$

Norm $G_z$ is $||Z g||$, the Euclidean norm of the projected gradient (see Section 8.1). Norm $G_z$ will be approximately zero in the neighbourhood of a solution.

Cond $H$ is a lower bound on the condition number of the Hessian approximation $H$.

Cond $Hz$ is a lower bound on the condition number of the projected Hessian approximation $H$ ($H = Z H Z = R R$; see (6) and (12) in Sections 3 and 8.1). The larger this number, the more difficult the problem.

Cond $T$ is a lower bound on the condition number of the matrix of predicted active constraints.

Norm $C$ is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if NCNLN is zero). Norm $C$ will be approximately zero in the neighbourhood of a solution.

Penalty is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if NCNLN is zero).

Conv is a three-letter indication of the status of the three convergence tests (8a)-(8c) defined in the description of the optional parameter Optimality Tolerance in Section 5.1. Each letter is T if the test is satisfied, and F otherwise. The three tests indicate whether:

(a) the sequence of iterates has converged;

(b) the projected gradient (Norm $G_z$) is sufficiently small; and

(c) the norm of the residuals of constraints in
the predicted active set (Norm C) is small enough.
If any of these indicators is F when E04UCF terminates with IFAIL = 0, the user should check the solution carefully.

M is printed if the Quasi-Newton update was modified to ensure that the Hessian approximation is positive-definite (see Section 8.3).

I is printed if the QP subproblem has no feasible point.

C is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of Step is zero, the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is resolved with the central-difference gradient and Jacobian.) If the value of Step is non-zero, central differences were computed because Norm Gz and Norm C imply that x is close to a Kuhn-Tucker point.

L is printed if the linesearch has produced a relative change in x greater than the value defined by the optional parameter Step Limit. If this output occurs frequently during later iterations of the run, Step Limit should be set to a larger value.

R is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned, the approximate Hessian is refactorized using column interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.

When Major Print Level = 1 or Major Print Level >= 10, the summary printout at the end of execution of E04UCF includes a listing of the status of every variable and constraint. Note that default names are assigned to all variables and constraints.

The following describes the printout for each variable.

Varbl gives the name (V) and index j=1,2,...,n of the variable.

State gives the state of the variable in the predicted
active set (FR if neither bound is in the active set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound). If the variable is predicted to lie outside its upper or lower bound by more than the feasibility tolerance, State will be ++ or -- respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)

Value is the value of the variable at the final iteration.

Lower bound is the lower bound specified for the variable. (None indicates that BL(j)<=-BIGND.)

Upper bound is the upper bound specified for the variable. (None indicates that BL(j)>=BIGND.)

Lagr Mult is the value of the Lagrange-multiplier for the associated bound constraint. This will be zero if State is FR. If x is optimal, the multiplier should be non-negative if State is LL, and non-positive if State is UL.

Residual is the difference between the variable Value and the nearer of its bounds BL(j) and BU(j).

The printout for general constraints is the same as for variables, except for the following:

L Con is the name (L) and index i, for i = 1,2,...,NCLIN of a linear constraint.

N Con is the name (N) and index i, for i = 1,2,...,NCNLN of a nonlinear constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

The input data for E04UCF should always be checked (even if E04UCF terminates with IFAIL=0).

Note that when Print Level>0, a short description of IFAIL is printed.

Errors and diagnostics indicated by IFAIL, together with some
recommendations for recovery are indicated below.

IFAIL = 1

The final iterate $x$ satisfies the first-order Kuhn-Tucker conditions to the accuracy requested, but the sequence of iterates has not yet converged. E04UCF was terminated because no further improvement could be made in the merit function.

This value of IFAIL may occur in several circumstances. The most common situation is that the user asks for a solution with accuracy that is not attainable with the given precision of the problem (as specified by Function Precision see Section 5). This condition will also occur if, by chance, an iterate is an 'exact' Kuhn-Tucker point, but the change in the variables was significant at the previous iteration. (This situation often happens when minimizing very simple functions, such as quadratics.)

If the four conditions listed in Section 5 for IFAIL = 0 are satisfied, $x$ is likely to be a solution of (1) even if IFAIL = 1.

IFAIL = 2

E04UCF has terminated without finding a feasible point for the linear constraints and bounds, which means that no feasible point exists for the given value of Linear Feasibility Tolerance (see Section 5.1). The user should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision (sigma), the user should ensure that the value of the optional parameter Linear Feasibility Tolerance is greater than ($\sigma$). For example, if all elements of $A$ are of order unity and are accurate to only three decimal places, Linear Feasibility Tolerance should be at least $10^{-3}$.

IFAIL = 3

No feasible point could be found for the nonlinear constraints. The problem may have no feasible solution. This means that there has been a sequence of QP subproblems for which no feasible point could be found (indicated by I at the end of each terse line of output). This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) If the infeasible subproblems occur from the very first major iteration, it is highly likely that no feasible point exists. If infeasibilities occur when earlier subproblems have been feasible, small constraint
inconsistencies may be present. The user should check the validity of constraints with negative values of ISTATE. If the user is convinced that a feasible point does exist, E04UCF should be restarted at a different starting point.

IFAIL= 4
The limiting number of iterations (determined by the optional parameter Major Iteration Limit see Section 5.1) has been reached.

If the algorithm appears to be making progress, Major Iteration Limit may be too small. If so, increase its value and rerun E04UCF (possibly using the Warm Start option). If the algorithm seems to be 'bogged down', the user should check for incorrect gradients or ill-conditioning as described below under IFAIL = 6.

Note that ill-conditioning in the working set is sometimes resolved automatically by the algorithm, in which case performing additional iterations may be helpful. However, ill-conditioning in the Hessian approximation tends to persist once it has begun, so that allowing additional iterations without altering R is usually inadvisable. If the quasi-Newton update of the Hessian approximation was modified during the latter iterations (i.e., an M occurs at the end of each terse line), it may be worthwhile to try a warm start at the final point as suggested above.

IFAIL= 6
x does not satisfy the first-order Kuhn-Tucker conditions, and no improved point for the merit function could be found during the final line search.

A sufficient decrease in the merit function could not be attained during the final line search. This sometimes occurs because an overly stringent accuracy has been requested, i.e., Optimality Tolerance is too small. In this case the user should apply the four tests described under IFAIL = 0 above to determine whether or not the final solution is acceptable (see Gill et al [10], for a discussion of the attainable accuracy).

If many iterations have occurred in which essentially no progress has been made and E04UCF has failed completely to move from the initial point then subroutines OBJFUN or CONFUN may be incorrect. The user should refer to comments below under IFAIL = 7 and check the gradients using the Verify parameter. Unfortunately, there may be small errors in the objective and constraint gradients that cannot be detected by the verification process. Finite-difference
approximations to first derivatives are catastrophically affected by even small inaccuracies. An indication of this situation is a dramatic alteration in the iterates if the finite-difference interval is altered. One might also suspect this type of error if a switch is made to central differences even when Norm Gz and Norm C are large.

Another possibility is that the search direction has become inaccurate because of ill-conditioning in the Hessian approximation or the matrix of constraints in the working set; either form of ill-conditioning tends to be reflected in large values of ItQP (the number of iterations required to solve each QP subproblem).

If the condition estimate of the projected Hessian (Cond Hz) is extremely large, it may be worthwhile to rerun E04UCF from the final point with the Warm Start option. In this situation, ISTATE should be left unaltered and R should be reset to the identity matrix.

If the matrix of constraints in the working set is ill-conditioned (i.e., Cond T is extremely large), it may be helpful to run E04UCF with a relaxed value of the Feasibility Tolerance (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed for Major Print Level >= 30).

IFAIL = 7
The user-provided derivatives of the objective function and/or nonlinear constraints appear to be incorrect.

Large errors were found in the derivatives of the objective function and/or nonlinear constraints. This value of IFAIL will occur if the verification process indicated that at least one gradient or Jacobian component had no correct figures. The user should refer to the printed output to determine which elements are suspected to be in error.

As a first-step, the user should check that the code for the objective and constraint values is correct - for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values x=0 or x=1 are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Special care should be used in this test if computation of the objective function involves subsidiary data communicated
in COMMON storage. Although the first evaluation of the function may be correct, subsequent calculations may be in error because some of the subsidiary data has accidently been overwritten.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single-precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

IFAIL= 9
An input parameter is invalid. The user should refer to the printed output to determine which parameter must be redefined.

IFAILOverflow
If the printed output before the overflow error contains a warning about serious ill-conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of the optional parameter Linear Feasibility Tolerance or Nonlinear Feasibility Tolerance, and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint (with index 'j') must be removed from the problem. If overflow occurs in one of the user-supplied routines (e.g. if the nonlinear functions involve exponentials or singularities), it may help to specify tighter bounds for some of the variables (i.e., reduce the gap between appropriate l  and u ).

7. Accuracy
If IFAIL = 0 on exit then the vector returned in the array X is an estimate of the solution to an accuracy of approximately Feasibility Tolerance (see Section 5.1), whose default value is 0.8 \( (\text{epsilon}) \), where \( (\text{epsilon}) \) is the machine precision (see X02AJF(*)).

8. Further Comments
In this section we give some further details of the method used
by E04UCF.

8.1. Solution of the Quadratic Programming Subproblem

The search direction \( p \) is obtained by solving (4) using the method of E04NCF(\( \ast \)) (Gill et al [8]), which was specifically designed to be used within an SQP algorithm for nonlinear programming.

The method of E04UCF is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the feasibility phase), and minimizing the quadratic objective function within the feasible region (the optimality phase). The computations in both phases are performed by the same subroutines. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let \( p \) denote the current estimate of the solution of (4); the new iterate \( p \) is defined by

\[
p = p + \sigma d,
\]

where, as in (3), \( \sigma \) is a non-negative step length and \( d \) is a search direction.

At the beginning of each iteration of E04UCF, a working set is defined of constraints (general and bound) that are satisfied exactly. The vector \( d \) is then constructed so that the values of constraints in the working set remain unaltered for any move along \( d \). For a bound constraint in the working set, this property is achieved by setting the corresponding component of \( d \) to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the components associated with the fixed and free variables.

Let \( C \) denote the sub-matrix of rows of

\[
\begin{pmatrix}
A \\
L \\
N
\end{pmatrix}
\]

corresponding to general constraints in the working set. The general constraints in the working set will remain unaltered if
\[ C \, d = 0, \quad (10) \]

which is equivalent to defining \( d \) as

\[ d = Zd \quad (11) \]

for some vector \( d \), where \( Z \) is the matrix associated with the TQ factorization (5) of \( C \). The definition of \( d \) in (11) depends on whether the current \( p \) is feasible. If not, \( d \) is zero except for a component \( (\gamma) \) in the \( j \)th position, where \( j \) and \( (\gamma) \) are chosen so that the sum of infeasibilities is decreasing along \( d \). (For further details, see Gill et al [8].) In the feasible case, \( d \) satisfies the equations

\[ \begin{aligned} R^T R \, d &= -Z^T q, \quad (12) \\
& \quad \text{z z z} \end{aligned} \]

where \( R \) is the Cholesky factor of \( Z^T Z \) and \( q \) is the gradient of the quadratic objective function \( (q = g + Hp) \). (The vector \( Z^T q \) is the projected gradient of the QP.) With (12), \( P + d \) is the minimizer of the quadratic objective function subject to treating the constraints in the working set as equalities.

If the QP projected gradient is zero, the current point is a constrained stationary point in the subspace defined by the working set. During the feasibility phase, the projected gradient will usually be zero only at a vertex (although it may vanish at non-vertices in the presence of constraint dependencies). During the optimality phase, a zero projected gradient implies that \( p \) minimizes the quadratic objective function when the constraints in the working set are treated as equalities. In either case, Lagrange multipliers are computed. Given a positive constant (\( \delta \)) of the order of the machine precision, the Lagrange multiplier (\( \mu \)) corresponding to an inequality constraint in the jth
working set at its upper bound is said to be optimal if 
\( (\mu) \leq (\delta) \) when the jth constraint is at its upper bound, or 
\( j \) if \( (\mu) \geq -(\delta) \) when the associated constraint is at its lower 
\( j \) bound. If any multiplier is non-optimal, the current objective 
function (either the true objective or the sum of 
ifeasibilities) can be reduced by deleting the corresponding 
constraint from the working set.

If optimal multipliers occur during the feasibility phase and the 
sum of infeasibilities is non-zero, no feasible point exists. The 
QP algorithm will then continue iterating to determine the 
minimum sum of infeasibilities. At this point, the Lagrange 
multiplier \( (\mu) \) will satisfy \( -(1+(\delta)) \leq (\mu) \leq (\delta) \) for an 
\( j \) inequality constraint at its upper bound, and 
\( -(\delta) \leq (\mu) \leq 1+(\delta) \) for an inequality at its lower bound. 
\( j \) The Lagrange multiplier for an equality constraint will satisfy 
\( |(\mu)| \leq 1+(\delta) \).
\( j \)

The choice of step length \( (\sigma) \) in the QP iteration (9) is 
based on remaining feasible with respect to the satisfied 
constraints. During the optimality phase, if \( p+d \) is feasible, 
\( (\sigma) \) will be taken as unity. (In this case, the projected 
gradient at \( p \) will be zero.) Otherwise, \( (\sigma) \) is set to 
\( (\sigma) \), the step to the 'nearest' constraint, which is added to 
the working set at the next iteration.

Each change in the working set leads to a simple change to \( C : 
\) if the status of a general constraint changes, a row of \( C \) is 
altered; if a bound constraint enters or leaves the working set, 
a column of \( C \) changes. Explicit representations are recurred of 
the matrices \( T, Q \) and \( R \), and of the vectors \( Q, q \) and \( Q, g \).

8.2. The Merit Function

After computing the search direction as described in Section 3, 
each major iteration proceeds by determining a step length 
\( (\alpha) \) in (3) that produces a 'sufficient decrease' in the 
augmented Lagrangian merit function.
\[ L(x, (\lambda), s) = F(x) - \sum_{i} \lambda_i (c_i(x) - s_i) \]
\[ + \sum_{i} \rho_i (c_i(x) - s_i), \quad (13) \]

where \( x, \lambda \) and \( s \) vary during the line search. The summation terms in (13) involve only the nonlinear constraints. The vector \( \lambda \) is an estimate of the Lagrange multipliers for the nonlinear constraints of (1). The non-negative slack variables \( \{s_i\} \) allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (4) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector \( \rho \) of penalty parameters is initialized to zero at the beginning of the first major iteration. Thereafter, selected components are increased whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of \( \rho \) (the printed quantity Penalty, see Section 5.2) is generally non-decreasing, although each \( \rho \) may be reduced a limited number of times. \[ i \]

The merit function (13) and its global convergence properties are described in Gill et al \[9\].

8.3. The Quasi-Newton Update

The matrix \( H \) in (4) is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel \[3\].) At the end of each major iteration, a new Hessian approximation \( H \) is defined as a rank-two modification of \( H \). In E04UCF, the BFGS quasi-Newton update is used:
\[
H = \frac{1}{s^T H s} H s s^T + \frac{1}{y^T y} y y^T , \quad (14)
\]

where \( s = x - x \) (the change in \( x \)).
In E04UCF, \( H \) is required to be positive-definite. If \( H \) is positive-definite, \( H \) defined by (14) will be positive-definite if and only if \( y^s \) is positive (see, e.g. Dennis and More [1]). Ideally, \( y \) in (14) would be taken as \( y \), the change in gradient of the Lagrangian function

\[
_\text{T} y = g-A(\mu) - g+A(\mu), \quad (15)
\]

where \( (\mu) \) denotes the QP multipliers associated with the nonlinear constraints of the original problem. If \( y^s \) is not sufficiently positive, an attempt is made to perform the update with a vector \( y \) of the form

\[
y = y + \gamma (\omega)(a^T(x)c(x) - a^T(x)c(x)), \quad \gamma > 0 \quad (\omega) \\
\]

where \( (\omega) \geq 0 \). If no such vector can be found, the update is performed with a scaled \( y \); in this case, \( M \) is printed to indicate that the update is modified.

Rather than modifying \( H \) itself, the Cholesky factor of the transformed Hessian \( \tilde{H} \) (6) is updated, where \( Q \) is the matrix from (5) associated with the active set of the QP subproblem. The update (13) is equivalent to the following update to \( H \):

\[
_\text{T} H = H - \left( \begin{array}{cc} 1 & \text{T} \end{array} \right) \begin{array}{cc} 1 & \text{T} \end{array} \tilde{H} \begin{array}{cc} 1 & \text{T} \end{array} H + \left( \begin{array}{cc} 1 & \text{T} \end{array} \right) y y, \quad (16)
\]

where \( y = Q y \), and \( s = Q s \). This update may be expressed as a rank-1 update of \( Q \).
one update to R (see Dennis and Schnabel [2]).

9. Example

This section describes one version of the so-called 'hexagon' problem (a different formulation is given as Problem 108 in Hock and Schittkowski [11]). The problem is to determine the hexagon of maximum area such that no two of its vertices are more than one unit apart (the solution is not a regular hexagon).

All constraint types are included (bounds, linear, nonlinear), and the Hessian of the Lagrangian function is not positive-definite at the solution. The problem has nine variables, non-infinite bounds on seven of the variables, four general linear constraints, and fourteen nonlinear constraints.

The objective function is

\[ F(x) = -x_2 x_6 + x_1 x_7 - x_3 x_7 + x_4 x_8 + x_5 x_9. \]

The bounds on the variables are

\[
\begin{align*}
  x_1 &\geq 0, \\
  -1 &\leq x_3 \leq 1, \\
  x_5 &\geq 0, \\
  x_6 &\geq 0, \\
  x_7 &\leq 0, \\
  x_8 &\leq 0.
\end{align*}
\]

Thus,

\[
\begin{align*}
  T_l &= (0, -\infty, -1, -\infty, 0, 0, 0, -\infty, -\infty) \\
  B_u &= (\infty, \infty, 1, \infty, \infty, \infty, \infty, 0, 0)
\end{align*}
\]

The general linear constraints are

\[
\begin{align*}
  x_2 - x_1 &\geq 0, \\
  x_3 - x_2 &\geq 0, \\
  x_4 - x_3 &\geq 0, \\
  x_5 - x_4 &\geq 0.
\end{align*}
\]

Hence,

\[
\begin{align*}
  l &= (0, A = (0 -1 0 0 0 0 0 0 0) \text{ and } u = (\infty, L = (0 0 0 1 -1 0 0 0 0) \text{ and } L = (\infty))
\end{align*}
\]

The nonlinear constraints are all of the form \( c_i(x) \leq 1, \) for \( i \)
i=1, 2, ..., 14; hence, all components of \( l \) are \(-\infty\), and all components of \( u \) are 1. The fourteen functions \( \{c(x)\} \) are

\[
\begin{align*}
2 & 2 \\
c(x) &= x + x, \\
1 & 1 6 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + (x - x), \\
2 & 2 1 7 6 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + x, \\
3 & 3 1 6 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + (x - x), \\
4 & 1 4 6 8 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + (x - x), \\
5 & 1 5 6 9 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= x + x, \\
6 & 2 7 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + x, \\
7 & 3 2 7 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + (x - x), \\
8 & 4 2 8 7 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + (x - x), \\
9 & 2 5 7 9 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + x, \\
10 & 4 3 8 \\
\end{align*}
\]

\[
\begin{align*}
2 & 2 \\
c(x) &= (x - x) + x, \\
11 & 5 3 9 \\
\end{align*}
\]
2274

CHAPTER 15. CHAPTER N

\[ c(x) = x + x, \]
\[ 12 \quad 4 \quad 8 \]

\[ c(x) = (x - x) + (x - x), \]
\[ 13 \quad 4 \quad 5 \quad 9 \quad 8 \]

\[ c(x) = x + x. \]
\[ 14 \quad 5 \quad 9 \]

An optimal solution (to five figures) is

* \[ x = (0.060947, 0.59765, 1.0, 0.59765, 0.060947, 0.34377, 0.5, -0.5, 0.34377), \]

* \[ F(x) = -1.34996. \] (The optimal objective function is unique, but is achieved for other values of \( x \).) Five nonlinear constraints and one simple bound are active at \( x \). The sample solution output is given later in this section, following the sample main program and problem definition.

Two calls are made to E04UCF in order to demonstrate some of its features. For the first call, the starting point is:

\[ x = (0.1, 0.125, 0.666666, 0.142857, 0.111111, 0.2, 0.25, -0.2, -0.25), \]

All objective and constraint derivatives are specified in the user-provided subroutines OBJFN1 and CONFN1, i.e., the default option Derivative Level = 3 is used.

On completion of the first call to E04UCF, the optimal variables are perturbed to produce the initial point for a second run in which the problem functions are defined by the subroutines OBJFN2 and CONFN2. To illustrate one of the finite-difference options in E04UCF, these routines are programmed so that the first six components of the objective gradient and the constant elements of the Jacobian matrix are not specified; hence, the option Derivative Level = 0 is chosen. During computation of the finite-difference intervals, the constant Jacobian elements are identified and set, and E04UCF automatically increases the derivative level to 2.
The second call to E04UCF illustrates the use of the Warm Start Level option to utilize the final active set, nonlinear multipliers and approximate Hessian from the first run. Note that Hessian = Yes was specified for the first run so that the array R would contain the Cholesky factor of the approximate Hessian of the Lagrangian.

The two calls to E04UCF illustrate the alternative methods of assigning default parameters. (There is no special significance in the order of these assignments; an options file may just as easily be used to modify parameters set by E04UEF.)

The results are typical of those obtained from E04UCF when solving well behaved (non-trivial) nonlinear problems. The approximate Hessian and working set remain relatively well-conditioned. Similarly the penalty parameters remain small and approximately constant. The numerical results illustrate much of the theoretically predicted behaviour of a quasi-Newton SQP method. As x approaches the solution, only one minor iteration is performed per major iteration, and the Norm Gz and Norm C columns exhibit the fast linear convergence rate mentioned in Sections 5 and 6. Note that the constraint violations converge earlier than the projected gradient. The final values of the project gradient norm and constraint norm reflect the limiting accuracy of the two quantities. It is possible to achieve almost full precision in the constraint norm but only half precision in the projected gradient norm. Note that the final accuracy in the nonlinear constraints is considerably better than the feasibility tolerance, because the constraint violations are being refined during the last few iterations while the algorithm is working to reduce the projected gradient norm. In this problem, the constraint values and Lagrange multipliers at the solution are 'well balanced', i.e., all the multipliers are approximately the same order of magnitude. The behaviour is typical of a well-scaled problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function E04UDF
E04UDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose
To supply optional parameters to E04UCF from an external file.

2. Specification

     SUBROUTINE E04UDF (IOPTNS, INFORM)
     INTEGER IOPTNS, INFORM

3. Description

E04UDF may be used to supply values for optional parameters to E04UCF. E04UDF reads an external file and each line of the file defines a single optional parameter. It is only necessary to supply values for those parameters whose values are to be different from their default values.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

     Print level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

The file containing the options must start with begin and must finish with end An example of a valid options file is:

     Begin * Example options file
     Print level =10
     End

Normally each line of the file is printed as it is read, on the
current advisory message unit (see X04ABF), but printing may be suppressed using the keyword nolist. To suppress printing of begin, nolist must be the first option supplied as in the file:

```
Begin
  Nolist
  Print level = 10
End
```

Printing will automatically be turned on again after a call to E04UCF and may be turned on again at any time by the user by using the keyword list.

Optional parameter settings are preserved following a call to E04UCF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the document for E04UCF.

4. References

None.

5. Parameters

1: IOPTNS -- INTEGER
   Input
   On entry: IOPTNS must be the unit number of the options file. Constraint: 0 ≤ IOPTNS ≤ 99.

2: INFORM -- INTEGER
   Output
   On exit: INFORM will be zero, if an options file with the current structure has been read. Otherwise INFORM will be positive. Positive values of INFORM indicate that an options file may not have been successfully read as follows:
   INFORM = 1
     IOPTNS is not in the range [0,99].

   INFORM = 2
     begin was found, but end-of-file was found before end was found.

   INFORM = 3
     end-of-file was found before begin was found.

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning
message is output on the current advisory message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UEF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function E04UEF
E04UEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

To supply individual optional parameters to E04UCF.

2. Specification

SUBROUTINE E04UEF (STRING)
CHARACTER*(*) STRING

3. Description

E04UEF may be used to supply values for optional parameters to E04UCF. It is only necessary to call E04UEF for those parameters whose values are to be different from their default values. One call to E04UEF sets one parameter value.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

Print level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following
items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77’s I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist. Thus the statement

    CALL E04UEF ('Nolist')

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04UCF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to E04UCF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

    CALL E04UEF ('Defaults')

prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the document for E04UCF.

4. References

None.

5. Parameters

1: STRING -- CHARACTER*(*) Input
   On entry: STRING must be a single valid option string. See Section 3 above and Section 5.1 of the routine document for E04UCF. On entry: STRING must be a single valid option
string. See Section 3 above and Section 5.1 of the routine
document for E04UCF.

6. Error Indicators and Warnings

If the parameter STRING is not recognised as a valid option
string, then a warning message is output on the current advisory
message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UDF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function

E04UCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E04YCF returns estimates of elements of the variance-covariance
matrix of the estimated regression coefficients for a nonlinear
least squares problem. The estimates are derived from the
Jacobian of the function f(x) at the solution.

This routine may be used following any one of the nonlinear
least-squares routines E04FCF(*), E04FDF, E04GBF(*), E04GCF,
E04GDF(*), E04GEF(*), E04HEF(*), E04HFF(*).

2. Specification

SUBRUTINE E04YCF (JOB, M, N, FSUMSQ, S, V, LV, CJ, WORK,
1 IFAIL)
INTEGER JOB, M, N, LV, IFAIL
DOUBLE PRECISION FSUMSQ, S(N), V(LV,N), CJ(N), WORK(N)

3. Description
E04YCF is intended for use when the nonlinear least-squares function, \( F(x) = f(x)^T f(x) \), represents the goodness of fit of a nonlinear model to observed data. The routine assumes that the Hessian of \( F(x) \), at the solution, can be adequately approximated by \( 2J^T J \), where \( J \) is the Jacobian of \( f(x) \) at the solution. The estimated variance-covariance matrix \( C \) is then given by

\[
2 \quad T^-1 \quad T
C = (\sigma) (J J) \quad J J \text{ non-singular},
\]

where \( (\sigma) \) is the estimated variance of the residual at the solution, \( x \), given by

\[
(\sigma) = \frac{2 F(x)}{m - n},
\]

\( m \) being the number of observations and \( n \) the number of variables.

The diagonal elements of \( C \) are estimates of the variances of the estimated regression coefficients. See the Chapter Introduction E04 and Bard [1] and Wolberg [2] for further information on the use of \( C \).

When \( J J \) is singular then \( C \) is taken to be

\[
2 \quad T^* \quad T
C = (\sigma) (J J) ,
\]

where \( (J J) \) is the pseudo-inverse of \( J J \), but in this case the parameter IFAIL is returned as non-zero as a warning to the user that \( J \) has linear dependencies in its columns. The assumed rank of \( J \) can be obtained from IFAIL.

The routine can be used to find either the diagonal elements of \( C \), or the elements of the \( j \)th column of \( C \), or the whole of \( C \).

E04YCF must be preceded by one of the nonlinear least-squares routines mentioned in Section 1, and requires the parameters FSUMSQ, \( S \) and \( V \) to be supplied by those routines. FSUMSQ is the
residual sum of squares \( F(x) \), and \( S \) and \( V \) contain the singular values and right singular vectors respectively in the singular value decomposition of \( J \). \( S \) and \( V \) are returned directly by the comprehensive routines E04FCF(*), E04GBF(*), E04GDF(*) and E04HEF(*), but are returned as part of the workspace parameter \( W \) from the easy-to-use routines E04FDF, E04GCF, E04GEF(*) and E04HFF(*). In the case of E04FDF, \( S \) starts at \( W(NS) \), where

\[
NS=6*N+2*M+M*N+1+\max(1,N*(N-1)/2)
\]

and in the cases of the remaining easy-to-use routines, \( S \) starts at \( W(NS) \), where

\[
NS=7*N+2*M+M*N+N*(N+1)/2+1+\max(1,N*(N-1)/2)
\]

The parameter \( V \) starts immediately following the elements of \( S \), so that \( V \) starts at \( W(NV) \), where

\[
NV=NS+N.
\]

For all the easy-to-use routines the parameter \( LV \) must be supplied as \( N \). Thus a call to E04YCF following E04FDF can be illustrated as

```fortran
CALL E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
NS = 6*N + 2*M + M*N + 1 + MAX((1,N*(N-1))/2)
NV = NS + N
CALL E04YCF (JOB, M, N, FSUMSQ, W(NS), W(NV), *
  N, CJ, WORK, IFAIL)
```

where the parameters \( M, N, FSUMSQ \) and the \((n+n)\) elements \( W(NS), W(NS+1),..., W(NV+N*N-1) \) must not be altered between the calls to E04FDF and E04YCF. The above illustration also holds for a call to E04YCF following a call to one of E04GCF, E04GEF(*), E04HFF(*) except that \( NS \) must be computed as

\[
NS = 7*N + 2*M + M*N + (N*(N+1))/2 + 1 + MAX((1,N*(N-1))/2)
\]

**4. References**


**5. Parameters**
1: JOB -- INTEGER Input
   On entry: which elements of C are returned as follows:
   JOB = -1
       The n by n symmetric matrix C is returned.
   JOB = 0
       The diagonal elements of C are returned.
   JOB > 0
       The elements of column JOB of C are returned.
   Constraint: -1 <= JOB <= N.

2: M -- INTEGER Input
   On entry: the number m of observations (residuals f(x)).
   Constraint: M >= N.

3: N -- INTEGER Input
   On entry: the number n of variables (x). Constraint: 1 <= N <= M.

4: FSUMSQ -- DOUBLE PRECISION Input
   On entry: the sum of squares of the residuals, F(x), at the
   solution x, as returned by the nonlinear least-squares
   routine. Constraint: FSUMSQ >= 0.0.

5: S(N) -- DOUBLE PRECISION array Input
   On entry: the n singular values of the Jacobian as returned
   by the nonlinear least-squares routine. See Section 3 for
   information on supplying S following one of the easy-to-use
   routines.

6: V(LV,N) -- DOUBLE PRECISION array Input/Output
   On entry: the n by n right-hand orthogonal matrix (the
   right singular vectors) of J as returned by the nonlinear
   least-squares routine. See Section 3 for information on
   supplying V following one of the easy-to-use routines. On
   exit: when JOB >= 0 then V is unchanged.
   When JOB = -1 then the leading n by n part of V is
   overwritten by the n by n matrix C. When E04YCF is called
   with JOB = -1 following an easy-to-use routine this means
   that C is returned, column by column, in the n elements of
2

W given by W(NV), W(NV+1), ..., W(NV+N -1). (See Section 3 for the definition of NV).

7: LV -- INTEGER  
Input
On entry:
the first dimension of the array V as declared in the
(sub)program from which E04YCF is called.
When V is passed in the workspace parameter W following one
of the easy-to-use least-square routines, LV must be the
value N.

8: CJ(N) -- DOUBLE PRECISION array  
Output
On exit: with JOB = 0, CJ returns the n diagonal elements
of C.

With JOB = j>0, CJ returns the n elements of the jth column
of C.

When JOB = -1, CJ is not referenced.

9: WORK(N) -- DOUBLE PRECISION array  
Workspace
When JOB = -1 or 0 then WORK is used as internal workspace.

When JOB > 0, WORK is not referenced.

10: IFAIL -- INTEGER  
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
the output of an error message when soft failure occurs, set
IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
On entry JOB < -1,

or  JOB > N,
or \( N < 1 \),
or \( M < N \),
or \( \text{FSUMSQ} < 0.0 \).

IFAIL= 2
The singular values are all zero, so that at the solution
the Jacobian matrix \( J \) has rank 0.

IFAIL> 2
At the solution the Jacobian matrix contains linear, or near
linear, dependencies amongst its columns. In this case the
required elements of \( C \) have still been computed based upon \( J \)
having an assumed rank given by \( (\text{IFAIL}-2) \). The rank is
computed by regarding singular values \( \text{SV}(j) \) that are not
larger than \( 10^*(\text{epsilon})*\text{SV}(1) \) as zero, where \( (\text{epsilon}) \) is
the machine precision (see \( \text{X02AJF}(*) \)). Users who expect near
linear dependencies at the solution and are happy with this
tolerance in determining rank should call \( \text{E04YCF} \) with IFAIL
= 1 in order to prevent termination by \( \text{P01ABF}(*) \). It is then
essential to test the value of IFAIL on exit from \( \text{E04YCF} \).

IFAILOverflow
If overflow occurs then either an element of \( C \) is very
large, or the singular values or singular vectors have been
incorrectly supplied.

7. Accuracy

The computed elements of \( C \) will be the exact covariances
corresponding to a closely neighbouring Jacobian matrix \( J \).

8. Further Comments

When \( \text{JOB} = -1 \) the time taken by the routine is approximately
\( \frac{3}{2} \) proportional to \( n \). When \( \text{JOB} \geq 0 \) the time taken by the routine
is approximately proportional to \( n \).

9. Example

To estimate the variance-covariance matrix \( C \) for the least-
squares estimates of \( x \), \( x \) and \( x \) in the model
\[
\begin{align*}
t & \\
1 & = x + ---------
\end{align*}
\]
using the 15 sets of data given in the following table:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
0.14 & 1.0 & 15.0 & 1.0 \\
0.18 & 2.0 & 14.0 & 2.0 \\
0.22 & 3.0 & 13.0 & 3.0 \\
0.25 & 4.0 & 12.0 & 4.0 \\
0.29 & 5.0 & 11.0 & 5.0 \\
0.32 & 6.0 & 10.0 & 6.0 \\
0.35 & 7.0 & 9.0 & 7.0 \\
0.39 & 8.0 & 8.0 & 8.0 \\
0.37 & 9.0 & 7.0 & 7.0 \\
0.58 & 10.0 & 6.0 & 6.0 \\
0.73 & 11.0 & 5.0 & 5.0 \\
0.96 & 12.0 & 4.0 & 4.0 \\
1.34 & 13.0 & 3.0 & 3.0 \\
2.10 & 14.0 & 2.0 & 2.0 \\
4.39 & 15.0 & 1.0 & 1.0 \\
\end{array}
\]

The program uses \((0.5, 1.0, 1.5)\) as the initial guess at the position of the minimum and computes the least-squares solution using E04FDF. See the routine document E04FDF for further information.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
PACKAGE NAGE04 NAGOPTIMISATIONPACKAGE

NagOptimisationPackage (NAGE04)

Exports:
- e04dgf
- e04fdf
- e04gcf
- e04jaf
- e04mbf
- e04naf
- e04ucf
- e04ycf

--- package NAGE04 NagOptimisationPackage ---

\verb+\verb+ package NAGE04 NagOptimisationPackage \verb+\verb+\n++ Author: Godfrey Nolan and Mike Dewar\n++ Date Created: Jan 1994\n++ Date Last Updated: Thu May 12 17:45:09 1994\n++ Description:\n++ This package uses the NAG Library to perform optimization.\n++ An optimization problem involves minimizing a function (called\n++ the objective function) of several variables, possibly subject to\n++ restrictions on the values of the variables defined by a set of\n++ constraint functions. The routines in the NAG Foundation Library\n++ are concerned with function minimization only, since the problem\n++ of maximizing a given function can be transformed into a\n++ minimization problem simply by multiplying the function by -1.\n
NagOptimisationPackage(): Exports == Implementation where\nS ==> Symbol\nFOP ==> FortranOutputStackPackage\n
Exports == with\ne04dgf : (Integer,DoubleFloat,DoubleFloat,Integer,\n  DoubleFloat,Boolean,DoubleFloat,DoubleFloat,Integer,Integer,Integer,\n  Integer,Matrix DoubleFloat,Integer,\n  Union(fn:FileName,fp:Asp49(OBJFUN))) -> Result\n  ++ e04dgf(n,es,fu,it,lin,list,ma,op,pr,sta,sto,ve,x,ifail,objcun)\n  ++ minimizes an unconstrained nonlinear function of several\n  ++ variables using a pre-conditioned, limited memory quasi-Newton\n  ++ conjugate gradient method. First derivatives are required. The\n  ++ routine is intended for use on large scale problems.\n  ++ See \downlink{Manual Page}{manpageXXe04dgf}.\n
e04fdf : (Integer,Integer,Integer,Integer,_,
   Matrix DoubleFloat,Integer,_,
   Union(fn:FileName,p:Asp50(LSFUN1)) -> Result
++ e04fdf(m,n,liw,lw,x,ifail,lsfun1)
++ is an easy-to-use algorithm for finding an unconstrained
++ minimum of a sum of squares of m nonlinear functions in n
++ variables (m>=n). No derivatives are required.
++ See \downlink{Manual Page}{manpageXXe04fdf}.
e04gcf : (Integer,Integer,Integer,Integer,_,
   Matrix DoubleFloat,Integer,_,
   Union(fn:FileName,p:Asp19(LSFUN2)) -> Result
++ e04gcf(m,n,liw,lw,x,ifail,lsfun2)
++ is an easy-to-use quasi-Newton algorithm for finding an
++ unconstrained minimum of m nonlinear
++ functions in n variables (m>=n). First derivatives are required.
++ See \downlink{Manual Page}{manpageXXe04gcf}.
e04jaf : (Integer,Integer,Integer,Integer,_,
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_,
   Union(fn:FileName,p:Asp24(FUNCT1)) -> Result
++ e04jaf(n,ibound,liw,lw,bl,bu,x,ifail,funct1)
++ is an easy-to-use quasi-Newton algorithm for finding a
++ minimum of a function F(x,x,...,x), subject to fixed upper and
++ 1 2 n
++ lower bounds of the independent variables x ,x,... ,x, using
++ 1 2 n
++ function values only.
++ See \downlink{Manual Page}{manpageXXe04jaf}.
e04mbf : (Integer,Integer,Integer,Integer,_,
   Integer,Integer,Integer,Integer,_,
   Matrix DoubleFloat,Matrix DoubleFloat,_,
   Matrix DoubleFloat,Integer,Integer,_,
   Matrix DoubleFloat,Integer) -> Result
++ e04mbf(itmax,msglvl,n,nclin,nctotl,nrowa,a,bl,bu,
++ cvec,linobj,liwork,lwork,x,ifail)
++ is an easy-to-use routine for solving linear programming
++ problems, or for finding a feasible point for such problems. It
++ is not intended for large sparse problems.
++ See \downlink{Manual Page}{manpageXXe04mbf}.
e04naf : (Integer,Integer,Integer,Integer,Integer,Integer,Integer,_,
   Integer,Integer,Integer,Integer,Integer,Integer,Integer,_,
   Matrix DoubleFloat,Matrix DoubleFloat,_,
   Matrix DoubleFloat,Matrix DoubleFloat,_,
   Matrix DoubleFloat,Matrix DoubleFloat,_,
   Matrix DoubleFloat,Matrix Integer,Integer,_,
   Union(fn:FileName,p:Asp20(QPHESS)) -> Result
++ e04naf(itmax,msglvl1,n,nclin,nctotl,nrowa,a,bl,bu,
++ cvec,featol,liwork,lwork,x,istate,ifail,qphess)
++ is a comprehensive
++ programming (QP) or linear programming (LP) problems. It is not
++ intended for large sparse problems.
++ See \downlink{Manual Page}{manpageXXe04naf}.
e04ucf : (Integer,Integer,Integer,Integer,Integer,Integer,_)
++ \[ \text{e04ucf}(n, nclin, ncnl, nrowa, nrowj, nrowr, a, bl, bu, liwork, lwork, sta,} \\
++ \text{cra, der, fea, fun, hes, infb, infs, linf, list, maji, majp, mini,} \\
++ \text{minp, mon, nonf, opt, ste, stao, stac, stoo, stoc, ve, istate, cjac,} \\
++ \text{clamda, r, x, ifail, confun, objfun) } \\
++ \text{is designed to minimize an arbitrary smooth function} \\
++ \text{subject to constraints on the} \\
++ \text{variables, linear constraints.} \\
++ \text{(E04UCF may be used for unconstrained, bound-constrained and} \\
++ \text{linearly constrained optimization.) The user must provide} \\
++ \text{subroutines that define the objective and constraint functions} \\
++ \text{and as many of their first partial derivatives as possible.} \\
++ \text{Unspecified derivatives are approximated by finite differences.} \\
++ \text{All matrices are treated as dense, and hence E04UCF is not} \\
++ \text{intended for large sparse problems.} \\
++ \text{See } \text{\downlink{Manual Page}{manpageXXe04ucf}.} \\
++ \text{e04ycf : (Integer, Integer, Integer, DoubleFloat,} \\
++ \text{Matrix DoubleFloat, Integer, Matrix DoubleFloat, Integer) -> Result} \\
++ \text{e04ycf(job, m, n, fsumsq, s, lv, v, ifail) } \\
++ \text{returns estimates of elements of the variance} \\
++ \text{matrix of the estimated regression coefficients for a nonlinear} \\
++ \text{least squares problem. The estimates are derived from the} \\
++ \text{Jacobian of the function } f(x) \text{ at the solution.} \\
++ \text{See } \text{\downlink{Manual Page}{manpageXXe04ycf}.}
e04dgf(nArg: Integer, esArg: DoubleFloat, fuArg: DoubleFloat, _
  itArg: Integer, linArg: DoubleFloat, listArg: Boolean, _
  maArg: DoubleFloat, opArg: DoubleFloat, prArg: Integer, _
  staArg: Integer, stoArg: Integer, veArg: Integer, _
  xArg: Matrix DoubleFloat, ifailArg: Integer, _
  objfunArg: Union(fn: FileName, fp: Asp49(OBJFUN))): Result ==
  pushFortranOutputStack(objfunFilename := aspFilename "objfun")$FOP
  if objfunArg case fn
    then outputAsFortran(objfunArg.fn)
    else outputAsFortran(objfunArg.fp)
  popFortranOutputStack()$FOP
  [(invokeNagman([objfunFilename]$Lisp, _
    "e04dgf", _
    ["n": S, "es": S, "fu": S, "it": S, "lin": S_,
      "list": S, "ma": S, "op": S, "pr": S, "sta": S_,
      "sto": S, "ve": S, "iter": S, "objf": S, "ifail": S_,
      "objfun": S, "objgrd": S, "x": S, "iwork": S, "work": S, "iuser": S_,
      "user": S]$Lisp, _
    ["iter": S, "objf": S, "objgrd": S, "iwork": S, "work": S, "iuser": S_,
      "user": S, "objfun": S]$Lisp, _
    ["double": S, "es": S, "fu": S, "lin": S, "ma": S_,
      "op": S, "objf": S, ["objgrd": S, "x": S]$Lisp, ["x": S, "n": S]$Lisp, _
      ["work": S, ["*: S, 13$Lisp, "n": S]$Lisp]$Lisp, ["user": S, "*: S]$Lisp_,
      ["objfun": S]$Lisp, _
      ["integer": S, "n": S, "it": S, "pr": S, "sta": S_,
      "sto": S, "ve": S, "iter": S, "ifail": S, ["iwork": S_,
      ["*: S, "n": S]$Lisp]$Lisp, ["iuser": S, "*: S]$Lisp]$Lisp_,
      ["logical": S, "list": S]$Lisp]$Lisp_, _
    ])$Lisp_]
  pretend List (Record(key: Symbol, entry: Any))$Result

e04fdf(mArg: Integer, nArg: Integer, liwArg: Integer, _
  lwArg: Integer, xArg: Matrix DoubleFloat, ifailArg: Integer, _
  lsfun1Arg: Union(fn: FileName, fp: Asp50(LSFUN1))): Result ==
  pushFortranOutputStack(lsfun1Filename := aspFilename "lsfun1")$FOP
  if lsfun1Arg case fn
    then outputAsFortran(lsfun1Arg.fn)
    else outputAsFortran(lsfun1Arg.fp)
  popFortranOutputStack()$FOP
  [(invokeNagman([lsfun1Filename]$Lisp, _
    "e04fdf", _
    ["m": S, "n": S, "liw": S, "lw": S, "fsumsq": S_,
      "ifail": S, "lsfun1": S, "w": S, "x": S, "iw": S]$Lisp, _
      ["fsumsq": S, "w": S, "iw": S, "lsfun1": S]$Lisp, _
      ["double": S, "fsumsq": S, ["w": S, "lw": S]$Lisp_]}$Lisp_)
  pretend List (Record(key: Symbol, entry: Any))$Result
PACKAGENAGE04NAGOPTIMISATIONPACKAGE

2291

\[ \text{e04gcf}(m\text{Arg}: \text{Integer}, n\text{Arg}: \text{Integer}, liw\text{Arg}: \text{Integer}, lw\text{Arg}: \text{Integer}, x\text{Arg}: \text{Matrix DoubleFloat}, ifail\text{Arg}: \text{Integer}, \]
\[ \text{lsfun2\text{Arg}: \text{Union(fn: FileName, fp: Asp19(LSFUN2))}}): \text{Result} = \]
\[ \text{pushFortranOutputStack}(\text{lsfun2Filename} := \text{aspFilename \"lsfun2\")$FOP} \]
\[ \text{if lsfun2Arg case fn} \]
\[ \text{then outputAsFortran}(\text{lsfun2Arg.fn}) \]
\[ \text{else outputAsFortran}(\text{lsfun2Arg.fp}) \]
\[ \text{popFortranOutputStack}()$FOP \]
\[ \text{[(invokeNagman([\text{lsfun2Filename}]$Lisp,} \]
\[ \text{\"e04gcf\",} \]
\[ \text{[\"m\": S, \"n\": S, \"liw\": S, \"lw\": S, \"fsumsq\": S,} \]
\[ \text{\"ifail\": S, \"lsfun2\": S, \"x\": S, \"iw\": S]$Lisp,} \]
\[ \text{[\"fsumsq\": S, \"w\": S, \"liw\": S, \"lw\": S]$Lisp,} \]
\[ \text{[\"double\": S, \"fsumsq\": S, \[\"w\": S, \"lw\": S]$Lisp,} \]
\[ \text{[\"x\": S, \"n\": S]$Lisp, \"lsfun2\": S]$Lisp,} \]
\[ \text{[\"integer\": S, \"m\": S, \"n\": S, \"liw\": S, \"lw\": S,} \]
\[ \text{\"fsumsq\": S, \"w\": S, \"liw\": S, \"lw\": S]$Lisp,} \]
\[ \text{[(\[mArg::\text{Any}, nArg::\text{Any}, liwArg::\text{Any}, lwArg::\text{Any},} \]
\[ \text{ifailArg::\text{Any}, xArg::\text{Any }}]_\]
\[ \text{@List Any}$Lisp)$Lisp)$Lisp) \]
\[ \text{pretend List (Record(key: Symbol, entry: Any)))$Result} \]

\[ \text{e04jaf}(n\text{Arg}: \text{Integer}, ibound\text{Arg}: \text{Integer}, liw\text{Arg}: \text{Integer}, lw\text{Arg}: \text{Integer}, bl\text{Arg}: \text{Matrix DoubleFloat}, bu\text{Arg}: \text{Matrix DoubleFloat},} \]
\[ \text{x\text{Arg}: \text{Matrix DoubleFloat}, ifail\text{Arg}: \text{Integer},} \]
\[ \text{funct1\text{Arg}: \text{Union(fn: FileName, fp: Asp24(FUNCT1))}}): \text{Result} = \]
\[ \text{pushFortranOutputStack}(\text{funct1Filename} := \text{aspFilename \"funct1\")$FOP} \]
\[ \text{if funct1Arg case fn} \]
\[ \text{then outputAsFortran(funct1Arg.fn)} \]
\[ \text{else outputAsFortran(funct1Arg.fp)} \]
\[ \text{popFortranOutputStack}()$FOP \]
\[ \text{[(invokeNagman([\text{funct1Filename}]$Lisp,} \]
\[ \text{\"e04jaf\",} \]
\[ \text{[\"n\": S, \"ibound\": S, \"liw\": S, \"lw\": S, \"f\": S,} \]
\[ \text{\"ifail\": S, \"funct1\": S, \"bl\": S, \"bu\": S, \"x\": S, \"iw\": S, \"w\": S}$Lisp,} \]
\[ \text{[\"f\": S, \"iw\": S, \"w\": S, \"funct1\": S]$Lisp,} \]
\[ \text{[\"double\": S, \"f\": S, \[\"bl\": S, \"n\": S]$Lisp,} \]
\[ \text{[\"bu":: S, \"n\": S]$Lisp, \[\"x\": S, \"n\": S]$Lisp,} \]
CHAPTER 15. \textit{CHAPTER N}

\begin{verbatim}
pretend List (Record(key:Symbol,entry:Any))
pretend List (Record(key:Symbol,entry:Any))
e04mbf(itmaxArg:Integer,msglvlArg:Integer,nArg:Integer,
   nclinArg:Integer,nctotlArg:Integer,nrowaArg:Integer,linobjArg:Boolean,
   liworkArg:Integer,lworkArg:Integer,xArg:Matrix DoubleFloat,
   ifailArg:Integer): Result ==
   ([invokeNagman(NIL$Lisp,_
     "e04mbf",_
     \[\text{["double":S,\text{["a":S,"nrowa":S,"n":S]$Lisp_},_
        ,\text{["bl":S,"nctotl":S]$Lisp_},\text{["bu":S,"nctotl":S]$Lisp_},_
        ,\text{["cvec":S,"n":S]$Lisp_},\text{["clamda":S,"nctotl":S]$Lisp_},_
        ,\text{["x":S,"n":S]$Lisp_},\text{["lwork":S,"lwork":S,$Lisp_},_
        ,\text{["istate":S,"nctotl":S]$Lisp_},\text{["ifail":S,$Lisp_}
     \text{["lwork":S,"lwork":S]$Lisp_},\text{["logical":S,"linobj":S]$Lisp_},\text{["ifail":S}$Lisp_},
     \text{[\text{[itmaxArg:Any,msglvlArg:Any,nArg:Any,nclinArg:Any,_
        liworkArg:Any,ifailArg:Any,aArg:Any,blArg:Any,buArg:Any,_
        cvecArg:Any,xArg:Any ]}$Lisp_},@List Any$Lisp_)
   pretend List (Record(key:Symbol,entry:Any)))$Result
\end{verbatim}

\begin{verbatim}
e04naf(itmaxArg:Integer,msglvlArg:Integer,nArg:Integer,
   nclinArg:Integer,nctotlArg:Integer,nrowaArg:Integer,nrowhArg:Integer,
   ncolhArg:Integer,bigbndArg:DoubleFloat,
   aArg:Matrix DoubleFloat,blArg:Matrix DoubleFloat,
   buArg:Matrix DoubleFloat,featolArg:Matrix DoubleFloat,
   hessArg:Matrix DoubleFloat,
   coldArg:Boolean,ippArg:Boolean,orthogArg:Boolean,
   liworkArg:Integer,lworkArg:Integer,xArg:Matrix DoubleFloat,
   istateArg:Matrix Integer,ifailArg:Integer,
\end{verbatim}

\begin{verbatim}
pretend List (Record(key:Symbol,entry:Any))
pretend List (Record(key:Symbol,entry:Any))
e04naf(itmaxArg:Integer,msglvlArg:Integer,nArg:Integer,
   nclinArg:Integer,nctotlArg:Integer,nrowaArg:Integer,nrowhArg:Integer,
   ncolhArg:Integer,bigbndArg:DoubleFloat,
   aArg:Matrix DoubleFloat,blArg:Matrix DoubleFloat,
   buArg:Matrix DoubleFloat,featolArg:Matrix DoubleFloat,
   hessArg:Matrix DoubleFloat,
   coldArg:Boolean,ippArg:Boolean,orthogArg:Boolean,
   liworkArg:Integer,lworkArg:Integer,xArg:Matrix DoubleFloat,
   istateArg:Matrix Integer,ifailArg:Integer,
\end{verbatim}
qphessArg: Union(fn: FileName, fp: Asp20(QPHESS)) -> Result ==
  pushFortranOutputStack(qphessFilename := aspFilename "qphess")$FOP
  if qphessArg case fn
    then outputAsFortran(qphessArg.fn)
    else outputAsFortran(qphessArg.fp)
  popFortranOutputStack()$FOP
  [[invokeNagman([qphessFilename]$Lisp, _
                "e04naf",_ 
                 "cvec"::S,"feato1"::S_,
                 "work"::S]$Lisp,_,
                 "qphess"::S]$Lisp_,
                  ["cvec"::S,"n"::S]$Lisp_,["feato1"::S,"nctotl"::S]$Lisp_,
                  ["clamda"::S,"nctotl"::S]$Lisp, ["x"::S,"n"::S]$Lisp_,
                  ["work"::S,"iwork"::S]$Lisp_,
                  ["qphess"::S]$Lisp_,
                  [["integer"::S,"itmax"::S,"msglvl"::S,"n"::S_,
                    ["ifail"::S,["iwork"::S,"livork"::S]$Lisp]$Lisp_,
                    ["logical"::S,"cold"::S,"lpp"::S,"orthog"::S]$Lisp_]
                  ]$Lisp_,
                  [[[itmaxArg::Any,msglvlArg::Any,nArg::Any,nclinArg::Any_,
                    nctotlArg::Any,Any,nrowArg::Any,nrowhArg::Any,ncolhArg::Any_,
                    bigbndArg::Any,coldArg::Any,lppArg::Any,orthArg::Any_,
                    livorkArg::Any,lvorArg::Any,ifailArg::Any,akArg::Any,blArg::Any_,
                    buArg::Any,cveArg::Any,featoArg::Any,hessArg::Any,xArg::Any_,
                    istateArg::Any])))@List Any]$Lisp]$Lisp_]
  pretend List (Record(key: Symbol, entry: Any))$Result

e04ucf(nArg: Integer, nclinArg: Integer, ncnlnArg: Integer, _
        nrowArg: Integer, nrowjArg: Integer, nrowhArg: Integer, _
        aArg: Matrix DoubleFloat, blArg: Matrix DoubleFloat, _
        buArg: Matrix DoubleFloat, _
        livorkArg: Integer, lworkArg: Integer, staArg: Boolean, _
        craArg: DoubleFloat, derArg: Integer, feaArg: DoubleFloat, _
        funArg: DoubleFloat, hesArg: Boolean, infbArg: DoubleFloat, _
        infAsArg: DoubleFloat, linfArg: DoubleFloat, lintArg: DoubleFloat, _
        listArg: Boolean, majiArg: Integer, majpArg: Integer, _
        miniArg: Integer, minpArg: Integer, monArg: Integer, _
        nonfArg: DoubleFloat, optArg: DoubleFloat, steArg: DoubleFloat, _
staoArg:Integer,stacArg:Integer,stooArg:Integer,_
stocArg:Integer,veArg:Integer,istateArg:Matrix Integer,_
cjacArg:Matrix DoubleFloat,clamdaArg:Matrix DoubleFloat,_
rArg:Matrix DoubleFloat,_
xArg:Matrix DoubleFloat,ifailArg:Integer,_
confunArg:Union(fn:FileName,fp:Asp55(CONFUN)),_  
objfunArg:Union(fn:FileName,fp:Asp49(OBJFUN))): Result ==
pushFortranOutputStack(confunFilename := aspFilename "confun")$FOP
if confunArg case fn
  then outputAsFortran(confunArg.fn)
else outputAsFortran(confunArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(objfunFilename := aspFilename "objfun")$FOP
if objfunArg case fn
  then outputAsFortran(objfunArg.fn)
else outputAsFortran(objfunArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([confunFilename,objfunFilename]$Lisp,_
  "e04ucf",_  
  ["work"::S,"iwork"::S]$Lisp,$Lisp _,
  ["ifail"::S,"iwork"::S,"iwork"::S]$Lisp,$Lisp _,
  ["iuser"::S,"iuser"::S]$Lisp,$Lisp_
e04ycf( jobArg: Integer, mArg: Integer, nArg: Integer, _
  fsumsqArg: DoubleFloat, sArg: Matrix DoubleFloat, lvArg: Integer, _
  vArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
  [invokeNagman(NIL$Lisp, _
    "e04ycf", _
    ["job": S, "m": S, "n": S, "fsumsq": S, "lv": S, _
      "ifail": S, "s": S, "cj": S, "v": S, "work": S]$Lisp, _
    ["cj": S, "work": S]$Lisp, _
    [["double": S, "fsumsq": S, ["s": S, "n": S]$Lisp, _
      ["cj": S, "n": S]$Lisp, ["v": S, "lv": S, "n": S]$Lisp, _
      ["work": S, "n": S]$Lisp]$Lisp]$Lisp, _
    ["integer": S, "job": S, "m": S, "n": S, "lv": S, _
      "ifail": S]$Lisp]$Lisp, _
  ]$Lisp.$Lisp._
  pretend List (Record(key: Symbol, entry: Any))$Result
package NAGD02 NagOrdinaryDifferentialEquationsPackage

--- NagOrdinaryDifferentialEquationsPackage.input ---

)set break resume
)sys rm -f NagOrdinaryDifferentialEquationsPackage.output
)spool NagOrdinaryDifferentialEquationsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagOrdinaryDifferentialEquationsPackage
--E 1

)spool
)lisp (bye)

--- NagOrdinaryDifferentialEquationsPackage.help ---

This package uses the NAG Library to calculate the numerical solution of ordinary differential equations. There are two main types of problem, those in which all boundary conditions are specified at one point (initial-value problems), and those in which the boundary conditions are distributed between two or more points (boundary-value problems and eigenvalue problems). Routines are available for initial-value problems, two-point boundary-value problems and Sturm-Liouville eigenvalue problems.

1. Scope of the Chapter

This chapter is concerned with the numerical solution of ordinary differential equations. There are two main types of problem, those in which all boundary conditions are specified at one point (initial-value problems), and those in which the boundary conditions are distributed between two or more points (boundary-value problems and eigenvalue problems). Routines are available for initial-value problems, two-point boundary-value problems and
Sturm-Liouville eigenvalue problems.

2. Background to the Problems

For most of the routines in this chapter a system of ordinary differential equations must be written in the form

\[ y'_i = f(x, y_1, y_2, \ldots, y_n), \]
\[ y'_2 = f(x, y_2, y_1, \ldots, y_n), \]
\[ \ldots \]
\[ y'_n = f(x, y_n, y_1, \ldots, y_{n-1}), \]

that is the system must be given in first-order form. The \( n \) dependent variables (also, the solution) \( y_1, y_2, \ldots, y_n \) are functions of the independent variable \( x \), and the differential equations give expressions for the first derivatives \( y'_i = \frac{dy_i}{dx} \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \). For a system of \( n \) first-order equations, \( n \) associated boundary conditions are usually required to define the solution.

A more general system may contain derivatives of higher order, but such systems can almost always be reduced to the first-order form by introducing new variables. For example, suppose we have the third-order equation

\[ 2 \]
\[ z'''' + xz'' + k(l - z') = 0. \]

We write \( y = z, y = z', y = z'' \), and the third order equation may then be written as the system of first-order equations

\[ y'_1 = y \]
\[ y'_2 = y \]
\[ y'_3 = -y y_2 - k(1 - y_1). \]
For this system $n = 3$ and we require 3 boundary conditions in order to define the solution. These conditions must specify values of the dependent variables at certain points. For example, we have an initial-value problem if the conditions are:

1. $y = 0$ at $x=0$
2. $y = 0$ at $x=0$
3. $y = 0.1$ at $x=0$.

These conditions would enable us to integrate the equations numerically from the point $x=0$ to some specified end-point. We have a boundary-value problem if the conditions are:

1. $y = 0$ at $x=0$
2. $y = 0$ at $x=0$
3. $y = 1$ at $x=10$.

These conditions would be sufficient to define a solution in the range $0 \leq x \leq 10$, but the problem could not be solved by direct integration (see Section 2.2). More general boundary conditions are permitted in the boundary-value case.

2.1. Initial-value Problems

To solve first-order systems, initial values of the dependent variables $y_i$, for $i=1,2,...,n$ must be supplied at a given point, $a$. Also a point, $b$, at which the values of the dependent variables are required, must be specified. The numerical solution is then obtained by a step-by-step calculation which approximates values of the variables $y_i$, for $i=1,2,...,n$ at finite intervals over the required range $[a,b]$. The routines in this chapter adjust the step length automatically to meet specified accuracy tolerances. Although the accuracy tests used are reliable over each step individually, in general an accuracy requirement cannot be guaranteed over a long range. For many problems there may be no serious accumulation of error, but for unstable systems small perturbations of the solution will often lead to rapid divergence.
of the calculated values from the true values. A simple check for stability is to carry out trial calculations with different tolerances; if the results differ appreciably the system is probably unstable. Over a short range, the difficulty may possibly be overcome by taking sufficiently small tolerances, but over a long range it may be better to try to reformulate the problem.

A special class of initial-value problems are those for which the solutions contain rapidly decaying transient terms. Such problems are called stiff; an alternative way of describing them is to say that certain eigenvalues of the Jacobian matrix \( \frac{\partial f}{\partial y} \) have large negative real parts when compared to others. These problems require special methods for efficient numerical solution; the methods designed for non-stiff problems when applied to stiff problems tend to be very slow, because they need small step lengths to avoid numerical instability. A full discussion is given in Hall and Watt [6] and a discussion of the methods for stiff problems is given in Berzins, Brankin and Gladwell [1].

2.2. Boundary-value Problems

A full discussion of the design of the methods and codes for boundary-value problems is given in Gladwell [4]. In general, a system of nonlinear differential equations with boundary conditions given at two or more points cannot be guaranteed to have a solution. The solution has to be determined iteratively (if it exists). Finite-difference equations are set up on a mesh of points and estimated values for the solution at the grid points are chosen. Using these estimated values as starting values a Newton iteration is used to solve the finite-difference equations. The accuracy of the solution is then improved by deferred corrections or the addition of points to the mesh or a combination of both. Good initial estimates of the solution may be required in some cases but results may be difficult to compute when the solution varies very rapidly over short ranges. A discussion is given in Chapters 9 and 11 of Gladwell and Sayers [5] and Chapter 4 of Childs et al [2].

2.3. Eigenvalue Problems

Sturm-Liouville problems of the form

\[ (p(x)y')' + q(x, \lambda)y = 0 \]

with appropriate boundary conditions given at two points, can be solved by a Scaled Pruefer method. In this method the differential equation is transformed to another which can be solved for a specified eigenvalue by a shooting method. A
2.6. References


3. Recommendations on Choice and Use of Routines

There are no routines which deal directly with COMPLEX equations. These may however be transformed to larger systems of real equations of the required form. Split each equation into its real and imaginary parts and solve for the real and imaginary parts of each component of the solution. Whilst this process doubles the size of the system and may not always be appropriate it does make available for use the full range of routines provided presently.

3.1. Initial-value Problems

For simple first-order problems with low accuracy requirements, that is problems on a short range of integration, with derivative functions $f$ which are inexpensive to calculate and where only a few correct figures are required, the best routines to use are likely to be the Runge-Kutta-Merson (RK) routines, D02BBF and D02BHF. For larger problems, over long ranges or with high accuracy requirements the variable-order, variable-step Adams
routine D02CJF should usually be preferred. For stiff equations, that is those with rapidly decaying transient solutions, the Backward Differentiation Formula (BDF) variable-order, variable-step routine D02EJF should be used.

There are four routines for initial-value problems, two of which use the Runge-Kutta-Merson method:

- **D02BBF** integrates a system of first order ordinary differential equations over a range with intermediate output and a choice of error control.
- **D02BHF** integrates a system of first order ordinary differential equations with a choice of error control until a position is determined where a function of the solution is zero.

one uses an Adams method:

- **D02CJF** combines the functionality of D02BBF and D02BHF.

and one uses a BDF method:

- **D02EJF** combines the functionality of D02BBF and D02BHF.

### 3.2. Boundary-value Problems

D02GAF may be used for simple boundary-value problems with assigned boundary values. The user may find that convergence is difficult to achieve using D02GAF since only specifying the unknown boundary values and the position of the finite-difference mesh is permitted. In such cases the user may use D02RAF which permits specification of an initial estimate for the solution at all mesh points and allows the calculation to be influenced in other ways too. D02RAF is designed to solve a general nonlinear two-point boundary value problem with nonlinear boundary conditions.

A routine, D02GBF, is also supplied specifically for the general linear two-point boundary-value problem written in a standard

The user is advised to use interpolation routines from the E01 Chapter to obtain solution values at points not on the final mesh.

### 3.3. Eigenvalue Problems

There is one general purpose routine for eigenvalue problems, D02KEF. It may be used to solve regular or singular second-order Sturm-Liouville problems on a finite or infinite range.
Discontinuous coefficient functions can be treated and eigenfunctions can be computed.

D02 -- Ordinary Differential Equations
Chapter D02

Ordinary Differential Equations

D02BBF  ODEs, IVP, Runge-Kutta-Merson method, over a range, intermediate output
D02BHF  ODEs, IVP, Runge-Kutta-Merson method, until function of solution is zero
D02CJF  ODEs, IVP, Adams method, until function of solution is zero, intermediate output
D02EJF  ODEs, stiff IVP, BDF method, until function of solution is zero, intermediate output
D02GAF  ODEs, boundary value problem, finite difference technique with deferred correction, simple nonlinear problem
D02GBF  ODEs, boundary value problem, finite difference technique with deferred correction, general linear problem
D02KEF  2nd order Sturm-Liouville problem, regular/singular system, finite/infinite range, eigenvalue and eigenfunction, user-specified break-points
D02RAF  ODEs, general nonlinear boundary value problem, finite difference technique with deferred correction, continuation facility

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.
1. Purpose

D02BBF integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method, and returns the solution at points specified by the user.

2. Specification

```fortran
SUBROUTINE D02BBF (X, XEND, M, N, Y, TOL, IRELAB, RESULT,
                  FCN, OUTPUT, W, IFAIL)
    INTEGER M, N, IRELAB, IFAIL
    DOUBLE PRECISION X, XEND, Y(N), TOL, W(N,7), RESULT(M,N)
    EXTERNAL FCN, OUTPUT
```

3. Description

The routine integrates a system of ordinary differential equations

\[ y' = f(x, y_1, y_2, \ldots, y_n) \quad i = 1, 2, \ldots, n \]

from \( x = X \) to \( x = XEND \) using a Merson form of the Runge-Kutta method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \), and the values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at a set of points specified by the user. This solution is obtained by quintic Hermite interpolation on solution values produced by the Runge-Kutta method.

The accuracy of the integration and, indirectly, the interpolation is controlled by the parameters TOL and IRELAB.

For a description of Runge-Kutta methods and their practical implementation see Hall and Watt [1].

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION \hspace{1cm} \text{Input/Output}
On entry: X must be set to the initial value of the independent variable x. On exit: XEND, unless an error has occurred, when it contains the value of x at the error.

2: XEND -- DOUBLE PRECISION
   Input
   On entry: the final value of the independent variable. If XEND < X on entry, integration will proceed in a negative direction.

3: M -- INTEGER
   Input
   On entry: the first dimension of the array RESULT. This will usually be equal to the number of points at which the solution is required.
   Constraint: M > 0.

4: N -- INTEGER
   Input
   On entry: the number of differential equations.
   Constraint: N > 0.

5: Y(N) -- DOUBLE PRECISION array
   Input/Output
   On entry: the initial values of the solution y_1, y_2, ..., y_N.
   On exit: the computed values of the solution at the final value of X.

6: TOL -- DOUBLE PRECISION
   Input/Output
   On entry: TOL must be set to a positive tolerance for controlling the error in the integration.
   Constraint: TOL > 0.0. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution then, on return, TOL has its sign changed. This should be treated as a warning that the computed solution is likely to be more accurate than would be produced by using the same value of TOL.
TOL on a longer range of integration.

7: IRELAB -- INTEGER
   Input
On entry: IRELAB determines the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:
IRELAB = 0
   \[ \text{EST} = 10.0 \leq \text{TOL} \times \max\{1.0, |y_1|, |y_2|, \ldots, |y_n|\} \]
IRELAB = 1
   \[ \text{EST} \leq \text{TOL} \]
IRELAB = 2
   \[ \text{EST} \leq \text{TOL} \times \max\{\text{epsilon}, |y_1|, |y_2|, \ldots, |y_n|\} \]
   where (epsilon) is machine precision.
If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step.

If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then IRELAB should be given the value 1 on entry, whereas if the error requirement is in terms of the number of correct significant digits, then IRELAB should be given the value 2. Where there is no preference in the choice of error test IRELAB = 0 will result in a mixed error test. Constraint: 0 \leq \text{IRELAB} \leq 2.

8: RESULT(M,N) -- DOUBLE PRECISION array
   Output
On exit: the computed values of the solution at the points given by OUTPUT.

9: FCN -- SUBROUTINE, supplied by the user.
   External Procedure
FCN must evaluate the functions f (i.e., the derivatives \( y'_i \)) for given values of its arguments \( x, y_1, \ldots, y_n \).

Its specification is:

\[
\text{SUBROUTINE FCN (X, Y, F)}
\]
\[
\text{DOUBLE PRECISION X, Y(n), F(n)}
\]

where n is the actual value of N in the call of D02BBF.

1: X -- DOUBLE PRECISION
   Input
On entry: the value of the argument x.

2: \( Y(*) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Input}
On entry: the value of the argument \( y \), for
\( i = 1, 2, \ldots, n. \)

3: \( F(*) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Output}
On exit: the value of \( f \), for \( i = 1, 2, \ldots, n. \)

FCN must be declared as EXTERNAL in the (sub)program
from which D02BBF is called. Parameters denoted as Input must not be changed by this procedure.

10: \( \text{OUTPUT} \) -- SUBROUTINE, supplied by the user.
\hspace{1cm} \text{External Procedure}
OUTPUT allows the user to have access to intermediate values
of the computed solution at successive points specified by the
user. These solution values may be returned to the user via
the array RESULT if desired (this is a non-standard feature
added for use with the AXIOM system). OUTPUT is initially
called by D02BBF with XSOL = X (the initial value of x). The
user must reset XSOL to the next point where OUTPUT is to be
called, and so on at each call to OUTPUT. If, after a call
to OUTPUT, the reset point XSOL is beyond XEND, D02BBF will
integrate to XEND with no further calls to OUTPUT; if a call
to OUTPUT is required at the point XSOL = XEND, then XSOL
must be given precisely the value XEND.

Its specification is:

\begin{verbatim}
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
INTEGER M,N,COUNT
1: XSOL -- DOUBLE PRECISION \hspace{1cm} \text{Input/Output}
On entry: the current value of the independent
variable x. On exit: the next value of x at which
OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array \hspace{1cm} \text{Input}
On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER \hspace{1cm} \text{Input/Output}
On entry: Zero if OUTPUT has not been called before, or
the previous value of COUNT.
On exit: A new value of COUNT: this can be used to keep
track of the number of times OUTPUT has been called.

4: M -- INTEGER \hspace{1cm} \text{Input}
\end{verbatim}
On entry: The first dimension of RESULT.

5: \( N \) -- INTEGER
    Input
    On entry: The dimension of \( Y \).

6: RESULT\( (M,N) \) -- DOUBLE PRECISION array
    Input/Output
    On entry: the previous contents of RESULT.
    On exit: RESULT may be used to return the values of the
    intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02BBF is called. Parameters denoted as
Input must not be changed by this procedure.

11: \( W(N,7) \) -- DOUBLE PRECISION array
    Workspace

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
On entry TOL \( \leq 0 \),
    or \( N \leq 0 \),
    or IRELAB \( /= 0, 1 \) or 2.

IFAIL = 2
With the given value of TOL, no further progress can be made
across the integration range from the current point \( x = X \),
or the dependence of the error on TOL would be lost if
further progress across the integration range were attempted
(see Section 8 for a discussion of this error exit). The
components \( Y(1),Y(2),\ldots,Y(n) \) contain the computed values of
the solution at the current point \( x = X \).

IFAIL = 3
TOL is too small for the routine to take an initial step
(see Section 8). \( X \) and \( Y(1),Y(2),\ldots,Y(n) \) retain their
initial values.

IFAIL = 4
X = XEND and XSOL /= X after the initial call to OUTPUT.

IFAIL= 5
A value of XSOL returned by OUTPUT lies behind the previous value of XSOL in the direction of integration.

IFAIL= 6
A serious error has occurred in an internal call to D02PAF(*). Check all subroutine calls and array dimensions. Seek expert help.

IFAIL= 7
A serious error has occurred in an internal call to D02XAF(*). Check all subroutine calls and array dimensions. Seek expert help.

7. Accuracy

The accuracy depends on TOL, on the mathematical properties of the differential system, on the length of the range of integration and on the method. It can be controlled by varying TOL but the approximate proportionality of the error to TOL holds only for a restricted range of values of TOL. For TOL too large, the underlying theory may break down and the result of varying TOL may be unpredictable. For TOL too small, rounding errors may affect the solution significantly and an error exit with IFAIL = 2 or IFAIL = 3 is possible.

At the intermediate output points the same remarks apply. For large values of TOL it is possible that the errors at some intermediate output points may be much larger than at XEND. In any case, it must not be expected that the error will have the same size at all output points. At any point, it is a combination of the errors arising from the integration of the differential equation and the interpolation. The effect of combining these errors will vary, though in most cases the integration error will dominate.

The user who requires a more reliable estimate of the accuracy achieved than can be obtained by varying TOL, is recommended to call D02BDF(*) where both the solution and a global error estimate are computed.

8. Further Comments

The time taken by the routine depends on the complexity and mathematical properties of the system of differential equations defined by FCN, on the range, the tolerance and the number of calls to OUTPUT. There is also an overhead of the form a+n where a and b are machine-dependent computing times.
If the routine fails with IFAIL = 3, then it can be called again with a larger value of TOL (if this has not already been tried). If the accuracy requested is really needed and cannot be obtained with this routine, the system may be very stiff (see below) or so badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has been called with a value of TOL which is so small that the solution cannot be obtained on the range X to XEND. This can happen for well-behaved systems and very small values of TOL. The user should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. If overflow occurs using D02BBF, D02PAF(*) can be used instead to trap the increasing solution before overflow occurs. In any case, numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations, where the solution contains rapidly decaying components, the routine will use very small steps in x (internally to D02BBF) to preserve stability. This will usually exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Merson's method is not efficient in such cases, and the user should try using D02EBF(*) (Backward Differentiation Formula). To determine whether a problem is stiff, D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness or singularities, the Merson method should work well for low accuracy calculations (three or four figures). For higher accuracy calculations or where FCN is costly to evaluate, Merson's method may not be appropriate and a computationally less expensive method may be D02CBF(*) which uses an Adams method.

Users with problems for which D02BBF is not sufficiently general should consider using D02PAF(*) with D02XAF(*). D02PAF(*) is a more general Merson routine with many facilities including more general error control options and several criteria for interrupting the calculations. D02XAF(*) interpolates on values produced by D02PAF(*).

9. Example

To integrate the following equations (for a projectile)
\[ y' = \tan(\phi) \]
\[ v' = \frac{-0.032\tan(\phi) + 0.02v}{\cos(\phi)} \]
\[ (\phi)' = \frac{-0.032}{v^2} \]

over an interval \( X = 0.0 \) to \( X_{\text{END}} = 8.0 \), starting with values \( y=0.0 \), \( v=0.5 \) and \( (\phi)=(\pi)/5 \) and printing the solution at steps of 1.0. We write \( y=Y(1) \), \( v=Y(2) \) and \( (\phi)=Y(3) \), and we set \( \text{TOL}=1.0\text{E}-4 \) and \( \text{TOL}=1.0\text{E}-5 \) in turn so that we may compare the solutions. The value of \( (\pi) \) is obtained by using \text{X01AAF}(*)

Note the careful construction of routine \text{OUT} to ensure that the value of \( X_{\text{END}} \) is printed.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

D02BHF(3NAG)  Foundation Library (12/10/92)  D02BHF(3NAG)

D02 -- Ordinary Differential Equations
D02BHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
D02BHF integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method, until a user-specified function of the solution is zero.

2. Specification

\[
\text{SUBROUTINE D02BHF} \ (X, \ XEND, \ N, \ Y, \ \text{TOL, IRELAB, HMAX, FCN, 1} \ G, \ W, \ IFAIL) \\
\text{INTEGER} \ \ N, \ \text{IRELAB, IFAIL} \\
\text{DOUBLE PRECISION} \ X, \ \text{XEND, Y(N), TOL, HMAX, G, W(N,7)}
\]
EXTERNAL FCN, G

3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y_i' = f(x, y_1, y_2, \ldots, y_n), \quad i = 1,2, \ldots, n, \]

from \( x = X \) towards \( x = X_{\text{END}} \) using a Merson form of the Runge-Kutta method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \) (see Section 5), and the values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

As the integration proceeds, a check is made on the function \( g(x,y) \) specified by the user, to determine an interval where it changes sign. The position of this sign change is then determined accurately by interpolating for the solution and its derivative. It is assumed that \( g(x,y) \) is a continuous function of the variables, so that a solution of \( g(x,y) = 0 \) can be determined by searching for a change in sign in \( g(x,y) \).

The accuracy of the integration and, indirectly, of the determination of the position where \( g(x,y) = 0 \), is controlled by the parameter TOL.

For a description of Runge-Kutta methods and their practical implementation see Hall and Watt [1].

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION  
   Input/Output  
   On entry: \( X \) must be set to the initial value of the independent variable \( x \). On exit: the point where \( g(x,y) = 0 \). 0 unless an error has occurred, when it contains the value of \( x \) at the error. In particular, if \( g(x,y) \neq 0.0 \) anywhere on the range \( X \) to \( X_{\text{END}} \), it will contain \( X_{\text{END}} \) on exit.

2: \( X_{\text{END}} \) -- DOUBLE PRECISION  
   Input  
   On entry: the final value of the independent variable \( x \).
If $XEND < X$ on entry, integration proceeds in a negative direction.

3: $N$ -- INTEGER  
   Input  
   On entry: the number of differential equations, $n$.  
   Constraint: $N > 0$.

4: $Y(N)$ -- DOUBLE PRECISION array  
   Input/Output  
   On entry: the initial values of the solution $y_1, y_2, \ldots, y_n$.  
   On exit: the computed values of the solution at the final point $x = X$.

5: $TOL$ -- DOUBLE PRECISION  
   Input/Output  
   On entry: $TOL$ must be set to a positive tolerance for controlling the error in the integration and in the determination of the position where $g(x,y) = 0.0$.  
   D02BHF has been designed so that, for most problems, a reduction in $TOL$ leads to an approximately proportional reduction in the error in the solution obtained in the integration. The relation between changes in $TOL$ and the error in the determination of the position where $g(x,y) = 0.0$ is less clear, but for $TOL$ small enough the error should be approximately proportional to $TOL$. However, the actual relation between $TOL$ and the accuracy cannot be guaranteed. The user is strongly recommended to call D02BHF with more than one value for $TOL$ and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge the user might compare results obtained by calling D02BHF with $TOL=10.0$ and $TOL=10.0^{-p}$ if $p$ correct decimal digits in the solution are required. Constraint: $TOL > 0.0$.  
   On exit: normally unchanged. However if the range from $x = X$ to the position where $g(x,y) = 0.0$ (or to the final value of $x$ if an error occurs) is so short that a small change in $TOL$ is unlikely to make any change in the computed solution, then $TOL$ is returned with its sign changed. To check results returned with $TOL < 0.0$, D02BHF should be called again with a positive value of $TOL$ whose magnitude is considerably smaller than that of the previous call.

6: $IRELAB$ -- INTEGER  
   Input  
   On entry: $IRELAB$ determines the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:  
   $IRELAB = 0$  
   $EST \leq TOL \times \max\{1.0, |y_1|, |y_2|, \ldots, |y_n|\}$;
IRELAB = 1
EST <= TOL;

IRELAB = 2
EST <= TOL*max\{\epsilon, |y_1|, |y_2|, \ldots, |y_n|\},

where (\epsilon) is machine precision.
If the appropriate condition is not satisfied, the step size
is reduced and the solution recomputed on the current step.

If the user wishes to measure the error in the computed
solution in terms of the number of correct decimal places,
then IRELAB should be given the value 1 on entry, whereas if
the error requirement is in terms of the number of correct
significant digits, then IRELAB should be given the value 2.
Where there is no preference in the choice of error test,
IRELAB = 0 will result in a mixed error test. It should be
borne in mind that the computed solution will be used in
evaluating \( g(x,y) \). Constraint: 0 <= IRELAB <= 2.

7: HMAX -- DOUBLE PRECISION Input
On entry: if HMAX = 0.0, no special action is taken.
If HMAX /= 0.0, a check is made for a change in sign of
\( g(x,y) \) at steps not greater than |HMAX|. This facility
should be used if there is any chance of 'missing' the
change in sign by checking too infrequently. For example, if
two changes of sign of \( g(x,y) \) are expected within a distance
h, say, of each other, then a suitable value for HMAX might
be HMAX = h/2. If only one change of sign in \( g(x,y) \) is
expected on the range X to XEND, then the choice HMAX = 0.0
is most appropriate.

8: FCN -- SUBROUTINE, supplied by the user.
External Procedure
FCN must evaluate the functions \( f \) (i.e., the derivatives
\( y' \) ) for given values of its arguments \( x, y_1, \ldots, y_n \).

SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)

1: X -- DOUBLE PRECISION Input
On entry: the value of the argument x.
2: Y(*) -- DOUBLE PRECISION array  
   On entry: the value of the argument \( y \), for \( i = 1, 2, \ldots, n \).

3: F(*) -- DOUBLE PRECISION array  
   On exit: the value of \( f \), for \( i = 1, 2, \ldots, n \).

FCN must be declared as EXTERNAL in the (sub)program from which D02BHF is called. Parameters denoted as Input must not be changed by this procedure.

9: G -- DOUBLE PRECISION FUNCTION, supplied by the user.  
   External Procedure
   G must evaluate the function \( g(x, y) \) at a specified point.

   Its specification is:

   \[
   \text{DOUBLE PRECISION FUNCTION G(X, Y)} \\
   \text{DOUBLE PRECISION X, Y(n)}
   \]

   where \( n \) is the actual value of \( N \) in the call of D02BHF.

1: X -- DOUBLE PRECISION  
   On entry: the value of the independent variable \( x \).

2: Y(*) -- DOUBLE PRECISION array  
   On entry: the value of \( y \), for \( i = 1, 2, \ldots, n \).

G must be declared as EXTERNAL in the (sub)program from which D02BHF is called. Parameters denoted as Input must not be changed by this procedure.

10: W(N,7) -- DOUBLE PRECISION array  
    Workspace

11: IFAIL -- INTEGER Input/Output  
    On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1  
On entry TOL <= 0.0,
or \( N \leq 0 \),
or \( \text{IRELAB} \neq 0, 1 \) or 2.

**IFAIL = 2**

With the given value of TOL, no further progress can be made
across the integration range from the current point \( x = X \),
or dependence of the error on TOL would be lost if further
progress across the integration range were attempted (see
Section 8 for a discussion of this error exit). The
components \( Y(1), Y(2), \ldots, Y(n) \) contain the computed values of
the solution at the current point \( x = X \). No point at which
\( g(x,y) \) changes sign has been located up to the point \( x = X \).

**IFAIL = 3**

TOL is too small for the routine to take an initial step
(see Section 8). \( X \) and \( Y(1), Y(2), \ldots, Y(n) \) retain their
initial values.

**IFAIL = 4**

At no point in the range \( X \) to \( X_{\text{END}} \) did the function \( g(x,y) \)
change sign. It is assumed that \( g(x,y) = 0.0 \) has no
solution.

**IFAIL = 5**

A serious error has occurred in an internal call to
\( \text{C05AZF}(\ast) \). Check all subroutine calls and array dimensions.
Seek expert help.

**IFAIL = 6**

A serious error has occurred in an internal call to
\( \text{D02PAF}(\ast) \). Check all subroutine calls and array dimensions.
Seek expert help.

**IFAIL = 7**

A serious error has occurred in an internal call to
\( \text{D02XAF}(\ast) \). Check all subroutine calls and array dimensions.
Seek expert help.

7. Accuracy

The accuracy depends on TOL, on the mathematical properties of
the differential system, on the position where \( g(x,y) = 0.0 \) and
on the method. It can be controlled by varying TOL but the
approximate proportionality of the error to TOL holds only for a
restricted range of values of TOL. For TOL too large, the
underlying theory may break down and the result of varying TOL
may be unpredictable. For TOL too small, rounding error may
affect the solution significantly and an error exit with IFAIL = 2 or IFAIL = 3 is possible.
The accuracy may also be restricted by the properties of \( g(x,y) \). The user should try to code \( G \) without introducing any unnecessary cancellation errors.

8. Further Comments

The time taken by the routine depends on the complexity and mathematical properties of the system of differential equations defined by FCN, the complexity of \( G \), on the range, the position of the solution and the tolerance. There is also an overhead of the form \( a+b*n \) where \( a \) and \( b \) are machine-dependent computing times.

For some problems it is possible that D02BHF will return IFAIL = 4 because of inaccuracy of the computed values \( Y \), leading to inaccuracy in the computed values of \( g(x,y) \) used in the search for the solution of \( g(x,y) = 0.0 \). This difficulty can be overcome by reducing TOL sufficiently, and if necessary, by choosing HMAX sufficiently small. If possible, the user should choose XEND well beyond the expected point where \( g(x,y) = 0.0 \); for example make |XEND-X| about 50 larger than the expected range. As a simple check, if, with XEND fixed, a change in TOL does not lead to a significant change in \( Y \) at XEND, then inaccuracy is not a likely source of error.

If the routine fails with IFAIL = 3, then it could be called again with a larger value of TOL if this has not already been tried. If the accuracy requested is really needed and cannot be obtained with this routine, the system may be very stiff (see below) or so badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is likely that it has been called with a value of TOL which is so small that a solution cannot be obtained on the range \( X \) to XEND. This can happen for well-behaved systems and very small values of TOL. The user should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. If overflow occurs using D02BHF, D02PAF(*) can be used instead to trap the increasing solution, before overflow occurs. In any case, numerical integration cannot be continued through a singularity, and analytical treatment should be considered;

(b) for 'stiff' equations, where the solution contains rapidly decaying components, the routine will compute in very small
steps in \( x \) (internally to D02BHF) to preserve stability. This will usually exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Merson's method is not efficient in such cases, and the user should try D02EHF(*) which uses a Backward Differentiation Formula method. To determine whether a problem is stiff, D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness or singularities, the Merson method should work well for low accuracy calculations (three or four figures). For high accuracy calculations or where FCN is costly to evaluate, Merson's method may not be appropriate and a computationally less expensive method may be D02CHF(*) which uses an Adams method.

For problems for which D02BHF is not sufficiently general, the user should consider D02PAF(*). D02PAF(*) is a more general Merson routine with many facilities including more general error control options and several criteria for interrupting the calculations. D02PAF(*) can be combined with the rootfinder C05AZF(*) and the interpolation routine D02XAF(*) to solve equations involving \( y_1, y_2, \ldots, y_n \) and their derivatives.

D02BHF can also be used to solve an equation involving \( x, y_1, y_2, \ldots, y_n \) and the derivatives of \( y_1, y_2, \ldots, y_n \). For example in Section 9, D02BHF is used to find a value of \( X > 0.0 \) where \( y(1) = 0.0 \). It could instead be used to find a turning-point of \( y_1 \) by replacing the function \( g(x,y) \) in the program by:

```fortran
DOUBLE PRECISION FUNCTION G(X,Y)
DOUBLE PRECISION X,Y(3),F(3)
CALL FCN(X,Y,F)
G = F(1)
RETURN
END
```

This routine is only intended to locate the first zero of \( g(x,y) \). If later zeros are required, users are strongly advised to construct their own more general root finding routines as discussed above.

9. Example

To find the value \( X > 0.0 \) at which \( y = 0.0 \), where \( y, v, (\phi) \) are defined by

\[
y' = \tan(\phi)
\]
\[-0.032\tan(\phi) \quad 0.02v\]
\[v' = \frac{-0.032}{v \cos(\phi)}\]
\[0.032\]
\[(\phi)' = -\frac{2}{v}\]

and where at \(X = 0.0\) we are given \(y=0.5\), \(v=0.5\) and \((\phi) = (\pi)/5\). We write \(y=Y(1)\), \(v=Y(2)\) and \((\phi)=Y(3)\) and we set \(TOL=1.0E-4\) and \(TOL=1.0E-5\) in turn so that we can compare the solutions. We expect the solution \(X \approx 7.3\) and so we set \(XEND = 10.0\) to avoid determining the solution of \(y=0.0\) too near the end of the range of integration. The value of \((\pi)\) is obtained by using \(X01AAF(*)\).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

**D02CJF(3NAG)**

**D02 -- Ordinary Differential Equations**

**D02CJF -- NAG Foundation Library Routine Document**

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.

1. Purpose

D02CJF integrates a system of first-order ordinary differential equations over a range with suitable initial conditions, using a variable-order, variable-step Adams method until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.

2. Specification

```fortran
SUBROUTINE D02CJF (X, XEND, M, N, Y, FCN, TOL, RELABS, RESULT, OUTPUT, G, W, IFAIL)
```
3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y' = f(x, y_1, y_2, \ldots, y_n), \quad i=1,2,\ldots,n, \]

from \( x = X \) to \( x = X_{\text{END}} \) using a variable-order, variable-step Adams method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \).

The initial values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at points specified by the user, if desired: this solution is obtained by C interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function \( g(x,y) \) to determine an interval where it changes sign. The position of this sign change is then determined accurately by C interpolation to the solution. It is assumed that \( g(x,y) \) is a continuous function of the variables, so that a solution of \( g(x,y)=0.0 \) can be determined by searching for a change in sign in \( g(x,y) \). The accuracy of the integration, the interpolation and, indirectly, of the determination of the position where \( g(x,y)=0.0 \), is controlled by the parameters TOL and RELABS.

For a description of Adams methods and their practical implementation see Hall and Watt [1].

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION Input/Output
   On entry: the initial value of the independent variable \( x \).
   Constraint: \( X \neq X_{\text{END}} \). On exit: if \( g \) is supplied by the user, it contains the point where \( g(x,y)=0.0 \), unless
g(x,y) /= 0.0 anywhere on the range X to XEND, in which case, X will contain XEND. If g is not supplied by the user it contains XEND, unless an error has occurred, when it contains the value of x at the error.

2: XEND -- DOUBLE PRECISION Input
On entry: the final value of the independent variable. If XEND < X, integration proceeds in the negative direction.
Constraint: XEND /= X.

3: M -- INTEGER Input
On entry: the first dimension of the array RESULT. This will usually be equal to the number of points at which the solution is required.
Constraint: M > 0.

4: N -- INTEGER Input
On entry: the number of differential equations.
Constraint: N >= 1.

5: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: the initial values of the solution y_i at x = X. On exit: the computed values of the solution at the final point x = X.

6: FCN -- SUBROUTINE, supplied by the user. External Procedure
FCN must evaluate the functions f (i.e., the derivatives y'_i) for given values of their arguments x, y_i, y_1, y_2, ..., y_n.

Its specification is:

SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)
where n is the actual value of N in the call of D02CJF.

1: X -- DOUBLE PRECISION Input
On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array Input
On entry: the value of the variable y_i, for i = 1, 2, ..., n.

3: F(*) -- DOUBLE PRECISION array Output
On exit: the value of f_i for i = 1, 2, ..., n.
FCN must be declared as EXTERNAL in the (sub)program from which D02CJF is called. Parameters denoted as Input must not be changed by this procedure.

7: TOL -- DOUBLE PRECISION

On entry: a positive tolerance for controlling the error in the integration. Hence TOL affects the determination of the position where \( g(x,y) = 0.0 \), if \( g \) is supplied.

D02CJF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02CJF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02CJF with TOL=10.0 and TOL=10.0 where \( p \) correct decimal digits are required in the solution.

Constraint: TOL > 0.0.

8: RELABS -- CHARACTER*1

On entry: the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

\[
\sqrt{n} \sum_{i=1}^{n} \left( \frac{e_i}{(\text{tau})_i \times |y_i| + (\text{tau})_i} \right) \leq 1.0
\]

where (\( \text{tau} \)) and (\( \text{tau} \)) are defined by

\[
\begin{align*}
\text{RELABS} & \ (\text{tau}) & (\text{tau}) \\
\text{'M'} & \ TOL & TOL \\
\text{'A'} & \ 0.0 & TOL \\
\text{'R'} & \ TOL & (\text{epsilon}) \\
\text{'D'} & \ TOL & TOL
\end{align*}
\]

where (\( \text{epsilon} \)) is a small machine-dependent number and \( e_i \) is an estimate of the local error at \( y_i \) computed internally. If the appropriate condition is not satisfied,
the step size is reduced and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then RELABS should be set to 'A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then RELABS should be set to 'R'. If the user prefers a mixed error test, then RELABS should be set to 'M', otherwise if the user has no preference, RELABS should be set to the default 'D'. Note that in this case 'D' is taken to be 'M'. Constraint: RELABS = 'M', 'A', 'R', 'D'.

9: RESULT(M,N) -- DOUBLE PRECISION array Output
On exit: the computed values of the solution at the points given by OUTPUT.

10: OUTPUT -- SUBROUTINE, supplied by the user.

External Procedure
OUTPUT allows the user to have access to intermediate values of the computed solution at successive points specified by the user. These solution values may be returned to the user via the array RESULT if desired (this is a non-standard feature added for use with the AXIOM system). OUTPUT is initially called by D02CJF with XSOL = X (the initial value of x). The user must reset XSOL to the next point where OUTPUT is to be called, and so on at each call to OUTPUT. If, after a call to OUTPUT, the reset point XSOL is beyond XEND, D02CJF will integrate to XEND with no further calls to OUTPUT; if a call to OUTPUT is required at the point XSOL = XEND, then XSOL must be given precisely the value XEND.

Its specification is:

```fortran
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
    DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
    INTEGER M,N,COUNT

1: XSOL -- DOUBLE PRECISION
    Input/Output
    On entry: the current value of the independent variable x. On exit: the next value of x at which OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array
    Input
    On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER
    Input/Output
    On entry: Zero if OUTPUT has not been called before, or the previous value of COUNT.
    On exit: A new value of COUNT: this can be used to keep track of the number of times OUTPUT has been called.
4:  M -- INTEGER                  Input
    On entry: The first dimension of RESULT.

5:  N -- INTEGER                  Input
    On entry: The dimension of Y.

6:  RESULT(M,N) -- DOUBLE PRECISION array Input/Output
    On entry: the previous contents of RESULT.
    On exit: RESULT may be used to return the values of
    the intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02CJF is called. Parameters denoted as
Input must not be changed by this procedure.

11: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
    External Procedure
    G must evaluate the function g(x,y) for specified values x,y
    . It specifies the function g for which the first position x
    where g(x,y)=0 is to be found.

If the user does not require the root finding option, the
actual argument G must be the dummy routine D02CJW. (D02CJW
is included in the NAG Foundation Library and so need not be
supplied by the user).

Its specification is:

    DOUBLE PRECISION FUNCTION G (X, Y)
    DOUBLE PRECISION X, Y(n)
    where n is the actual value of N in the call of D02CJF.

1:  X -- DOUBLE PRECISION        Input
    On entry: the value of the independent variable x.

2:  Y(*) -- DOUBLE PRECISION array      Input
    On entry: the value of the variable y, for
    i=1,2,...,n.

G must be declared as EXTERNAL in the (sub)program from
which D02CJF is called. Parameters denoted as Input
must not be changed by this procedure.

12: W(28+21*N) -- DOUBLE PRECISION array  Workspace

13: IFAIL -- INTEGER                  Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry TOL <= 0.0,
or  N <= 0,
or  RELABS /= 'M', 'A', 'R' or 'D'.
or  X = XEND.

IFAIL= 2
With the given value of TOL, no further progress can be made across the integration range from the current point x = X. (See Section 8 for a discussion of this error exit.) The components Y(1),Y(2),...,Y(N) contain the computed values of the solution at the current point x = X. If the user has supplied g, then no point at which g(x,y) changes sign has been located up to the point x = X.

IFAIL= 3
TOL is too small for D02CJF to take an initial step. X and Y(1),Y(2),...,Y(N) retain their initial values.

IFAIL= 4
XSOL has not been reset or XSOL lies behind X in the direction of integration, after the initial call to OUTPUT, if the OUTPUT option was selected.

IFAIL= 5
A value of XSOL returned by OUTPUT has not been reset or lies behind the last value of XSOL in the direction of integration, if the OUTPUT option was selected.

IFAIL= 6
At no point in the range X to XEND did the function g(x,y) change sign, if g was supplied. It is assumed that g(x,y)=0 has no solution.

IFAIL= 7
A serious error has occurred in an internal call. Check all
subroutine calls and array sizes. Seek expert help.

7. Accuracy

The accuracy of the computation of the solution vector Y may be controlled by varying the local error tolerance TOL. In general, a decrease in local error tolerance should lead to an increase in accuracy. Users are advised to choose RELABS = 'M' unless they have a good reason for a different choice.

If the problem is a root-finding one, then the accuracy of the root determined will depend on the properties of g(x,y). The user should try to code G without introducing any unnecessary cancellation errors.

8. Further Comments

If more than one root is required then D02QFF(*) should be used.

If the routine fails with IFAIL = 3, then it can be called again with a larger value of TOL if this has not already been tried. If the accuracy requested is really needed and cannot be obtained with this routine, the system may be very stiff (see below) or so badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has been called with a value of TOL which is so small that a solution cannot be obtained on the range X to XEND. This can happen for well-behaved systems and very small values of TOL. The user should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. Numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations where the solution contains rapidly decaying components, the routine will use very small steps in x (internally to D02CJF) to preserve stability. This will exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Adams methods are not efficient in such cases, and the user should try D02EJF.

9. Example

We illustrate the solution of four different problems. In each case the differential system (for a projectile) is
\[ y' = \tan(\phi) \]
\[ v' = \frac{-0.032 \tan(\phi) - 0.02v}{v \cos(\phi)} \]
\[ (\phi)' = \frac{-0.032}{2} \]

over an interval \( X = 0.0 \) to \( X_{END} = 10.0 \) starting with values \( y=0.5, v=0.5 \) and \( (\phi)=(\pi)/5 \). We solve each of the following problems with local error tolerances \( 1.0E-4 \) and \( 1.0E-5 \).

(i) To integrate to \( x=10.0 \) producing output at intervals of 2.0 until a point is encountered where \( y=0.0 \).

(ii) As (i) but with no intermediate output.

(iii) As (i) but with no termination on a root-finding condition.

(iv) As (i) but with no intermediate output and no root-finding termination condition.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.

2. Specification

SUBROUTINE D02EJF (X, XEND, M, N, Y, FCN, PEDERV, TOL, RELABS, OUTPUT, G, W, IW, RESULT, IFAIL)

INTEGER M, N, IW, IFAIL
DOUBLE PRECISION X, XEND, Y(N), TOL, G, W(IW), RESULT(M,N)
CHARACTER*1 RELABS
EXTERNAL FCN, PEDERV, OUTPUT, G

3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y' = f(x, y_1, y_2, \ldots, y_n), \quad i=1,2,\ldots,n, \]

from \( x = X \) to \( x = XEND \) using a variable-order, variable-step method implementing the BDF. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \) (see Section 5). The initial values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at points specified by the user, if desired: this solution is obtained by C interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function \( g(x, y) \) to determine an interval where it changes sign. The position of this sign change is then determined accurately by C interpolation to the solution. It is assumed that \( g(x,y) \) is a continuous function of the variables, so that a solution of \( g(x,y) = 0.0 \) can be determined by searching for a change in sign in \( g(x,y) \). The accuracy of the integration, the interpolation and, indirectly, of the determination of the position where \( g(x,y) = 0.0 \), is controlled by the parameters TOL and RELABS. The Jacobian of the system \( y' = f(x,y) \) may be supplied in routine PEDERV, if it is available.

For a description of BDF and their practical implementation see
4. References


5. Parameters

1: X -- DOUBLE PRECISION  
   Input/Output  
   On entry: the initial value of the independent variable x.  
   Constraint: X /= XEND  
   On exit: if G is supplied by the user, X contains the point where g(x,y) = 0.0, unless g(x,y) /= 0.  
   0 anywhere on the range X to XEND, in which case, X will contain XEND. If G is not supplied X contains XEND, unless  
   an error has occurred, when it contains the value of x at the  
   error.

2: XEND -- DOUBLE PRECISION  
   Input  
   On entry: the final value of the independent variable. If  
   XEND < X, integration proceeds in the negative direction.  
   Constraint: XEND /= X.

3: M -- INTEGER  
   Input  
   On entry: the first dimension of the array RESULT. This  
   will usually be equal to the number of points at which the  
   solution is required.  
   Constraint: M > 0.

4: N -- INTEGER  
   Input  
   On entry: the number of differential equations, n.  
   Constraint: N >= 1.

5: Y(N) -- DOUBLE PRECISION array  
   Input/Output  
   On entry: the initial values of the solution y_1, y_2, ..., y_n  
   at x = X. On exit: the computed values of the solution at  
   the final point x = X.

6: FCN -- SUBROUTINE, supplied by the user.  
   External Procedure  
   FCN must evaluate the functions f (i.e., the derivatives  
   y'_i) for given values of their arguments x, y_1, y_2, ..., y_n.  

   Its specification is:

   SUBROUTINE FCN (X, Y, F)  
   DOUBLE PRECISION X, Y(n), F(n)
where \( n \) is the actual value of \( N \) in the call of D02EJF.

1: \( X \) -- DOUBLE PRECISION
   Input
   On entry: the value of the independent variable \( x \).

2: \( Y(*) \) -- DOUBLE PRECISION array
   Input
   On entry: the value of the variable \( y \), for \( i = 1, 2, \ldots, n \).

3: \( F(*) \) -- DOUBLE PRECISION array
   Output
   On exit: the value of \( f \), for \( i = 1, 2, \ldots, n \).

FCN must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

7: PEDERV -- SUBROUTINE, supplied by the user.

External Procedure

PEDERV must evaluate the Jacobian of the system (that is, \( \frac{df}{dy} \)) for given values of the variables \( x, y_1, y_2, \ldots, y_n \).

Its specification is:

```fortran
SUBROUTINE PEDERV (X, Y, PW)
  DOUBLE PRECISION X, Y(n), PW(n,n)
```

where \( n \) is the actual value of \( N \) in the call of D02EJF.

1: \( X \) -- DOUBLE PRECISION
   Input
   On entry: the value of the independent variable \( x \).

2: \( Y(*) \) -- DOUBLE PRECISION array
   Input
   On entry: the value of the variable \( y \), for \( i = 1, 2, \ldots, n \).

3: \( PW(n,*) \) -- DOUBLE PRECISION array
   Output
   \( \frac{df}{dy} \)
   On exit: the value of \( \frac{df}{dy} \), for \( i,j = 1, 2, \ldots, n \).

If the user does not wish to supply the Jacobian, the actual argument PEDERV must be the dummy routine D02EJY
. (D02EJY is included in the NAG Foundation Library and so need not be supplied by the user. The name may be implementation dependent: see the User's Note for your implementation for details).

PEDERV must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

8: TOL -- DOUBLE PRECISION     Input/Output

On entry: TOL must be set to a positive tolerance for controlling the error in the integration. Hence TOL affects the determination of the position where $g(x,y) = 0.0$, if $G$ is supplied.

D02EJF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02EJF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02EJF with $TOL=10^p$ and $TOL=10^{p-1}$ if $p$ correct decimal digits are required in the solution. Constraint: $TOL > 0.0$. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution, then, on return, TOL has its sign changed.

9: RELABS -- CHARACTER*1     Input

On entry: the type of error control. At each step in the numerical solution an estimate of the local error, $EST$, is made. For the current step to be accepted the following condition must be satisfied:

\[
\begin{align*}
\text{EST} & = \left( e \left/ \left( \tau \right) \right. \right) \leq 1.0 \\
\text{where } \tau & \text{ and } \tau a \text{ are defined by }
\end{align*}
\]

where $\tau$ and $\tau a$ are defined by

\[
\begin{align*}
\text{RELABS} & (\tau) \quad (\tau a) \\
\text{'M'} & \quad TOL \quad TOL \\
\text{'A'} & \quad 0.0 \quad TOL
\end{align*}
\]
'R' \hspace{1em} TOL \hspace{1em} (\epsilon)

'D' \hspace{1em} TOL \hspace{1em} (\epsilon)

where \(\epsilon\) is a small machine-dependent number and \(e_i\)

is an estimate of the local error at \(y_i\), computed

internally. If the appropriate condition is not satisfied,

the step size is reduced and the solution is recomputed on

the current step. If the user wishes to measure the error in

the computed solution in terms of the number of correct
decimal places, then RELABS should be set to 'A' on entry,

whereas if the error requirement is in terms of the number

of correct significant digits, then RELABS should be set to

'R'. If the user prefers a mixed error test, then RELABS

should be set to 'M', otherwise if the user has no

preference, RELABS should be set to the default 'D'. Note

that in this case 'D' is taken to be 'R'. Constraint: RELABS

= 'A', 'M', 'R' or 'D'.

10: OUTPUT -- SUBROUTINE, supplied by the user.

External Procedure

OUTPUT allows the user to have access to intermediate values

of the computed solution at successive points specified by the

user. These solution values may be returned to the user via

the array RESULT if desired (this is a non-standard feature

added for use with the AXIOM system). OUTPUT is initially
called by D02EJF with XSOL = X (the initial value of x). The
user must reset XSOL to the next point where OUTPUT is to be
called, and so on at each call to OUTPUT. If, after a call
to OUTPUT, the reset point XSOL is beyond XEND, D02EJF will
integrate to XEND with no further calls to OUTPUT; if a call
to OUTPUT is required at the point XSOL = XEND, then XSOL
must be given precisely the value XEND.

Its specification is:

\begin{verbatim}
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
INTEGER M,N,COUNT

1: XSOL -- DOUBLE PRECISION Input/Output
   On entry: the current value of the independent
   variable x. On exit: the next value of x at which
   OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array Input
   On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER Input/Output
\end{verbatim}
On entry: Zero if OUTPUT has not been called before, or the previous value of COUNT.
On exit: A new value of COUNT: this can be used to keep track of the number of times OUTPUT has been called.

4: M -- INTEGER Input
On entry: The first dimension of RESULT.

5: N -- INTEGER Input
On entry: The dimension of Y.

6: RESULT(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the previous contents of RESULT.
On exit: RESULT may be used to return the values of the intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

11: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure
G must evaluate the function g(x,y) for specified values x,y. It specifies the function g for which the first position x where g(x,y) = 0 is to be found.

Its specification is:

DOUBLE PRECISION FUNCTION G (X, Y)
DOUBLE PRECISION X, Y(n)
where n is the actual value of N in the call of D02EJF.

1: X -- DOUBLE PRECISION Input
On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array Input
On entry: the value of the variable y, for
\[ i = 1,2,...,n. \]
If the user does not require the root finding option, the actual argument G must be the dummy routine D02EJW. (D02EJW is included in the NAG Foundation Library and so need not be supplied by the user).
G must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

12: W(IW) -- DOUBLE PRECISION array Workspace

13: IW -- INTEGER Input
On entry:
the dimension of the array W as declared in the (sub)program
from which D02EJF is called.
Constraint: IW>=(12+N)*N+50.

14: RESULT(M,N) -- DOUBLE PRECISION array       Output
On exit: the computed values of the solution at the points
given by OUTPUT.

15: IFAIL -- INTEGER                  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry TOL <= 0.0,
or          X = XEND,
or          N <= 0,
or          RELABS /= 'M', 'A', 'R', 'D'.
or          IW<(12+N)*N+50.

IFAIL= 2
With the given value of TOL, no further progress can be made
across the integration range from the current point x = X.
(See Section 5 for a discussion of this error test.) The
components Y(1),Y(2),...,Y(n) contain the computed values of
the solution at the current point x = X. If the user has
supplied G, then no point at which g(x,y) changes sign has
been located up to the point x = X.

IFAIL= 3
TOL is too small for D02EJF to take an initial step. X and Y
(1),Y(2),...,Y(n) retain their initial values.

IFAIL= 4
XSOL lies behind X in the direction of integration, after
the initial call to OUTPUT, if the OUTPUT option was selected.

IFAIL= 5
A value of XSOL returned by OUTPUT lies behind the last value of XSOL in the direction of integration, if the OUTPUT option was selected.

IFAIL= 6
At no point in the range X to XEND did the function g(x,y) change sign, if G was supplied. It is assumed that g(x,y) = 0 has no solution.

IFAIL= 7
A serious error has occurred in an internal call to C05AZF(*). Check all subroutine calls and array dimensions. Seek expert help.

IFAIL= 8
A serious error has occurred in an internal call to D02XKF(*). Check all subroutine calls and array dimensions. Seek expert help.

IFAIL= 9
A serious error has occurred in an internal call to D02NMF(*). Check all subroutine calls and array dimensions. Seek expert help.

7. Accuracy

The accuracy of the computation of the solution vector Y may be controlled by varying the local error tolerance TOL. In general, a decrease in local error tolerance should lead to an increase in accuracy. Users are advised to choose RELABS = 'R' unless they have a good reason for a different choice. It is particularly appropriate if the solution decays.

If the problem is a root-finding one, then the accuracy of the root determined will depend strongly on \( \frac{ddg}{ddx} \) and \( \frac{ddg}{ddy} \), for \( i = 1, 2, \ldots, n \). Large values for these quantities may imply large errors in the root.

8. Further Comments

If more than one root is required, then to determine the second and later roots D02EJF may be called again starting a short distance past the previously determined roots. Alternatively the
user may construct his own root finding code using D02QDF(*) (or the routines of the subchapter D02M-D02N), D02XKF(*) and C05AZF(*).

If it is easy to code, the user should supply the routine PEDERV. However, it is important to be aware that if PEDERV is coded incorrectly, a very inefficient integration may result and possibly even a failure to complete the integration (IFAIL = 2).

9. Example

We illustrate the solution of five different problems. In each case the differential system is the well-known stiff Robertson problem.

\[
\begin{align*}
4 & \quad a' = -0.04a - 10 bc \\
4 & \quad b' = 0.04a - 10 bc - 3*10 b \\
7 & \quad c' = 3*10 b
\end{align*}
\]

with initial conditions \(a=1.0, b=c=0.0\) at \(x=0.0\). We solve each of the following problems with local error tolerances 1.0E-3 and 1.0E-4.

(i) To integrate to \(x=10.0\) producing output at intervals of 2.0 until a point is encountered where \(a=0.9\). The Jacobian is calculated numerically.

(ii) As (i) but with the Jacobian calculated analytically.

(iii) As (i) but with no intermediate output.

(iv) As (i) but with no termination on a root-finding condition.

(v) Integrating the equations as in (i) but with no intermediate output and no root-finding termination condition.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
CHAPTER 15. CHAPTER N

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02GAF solves the two-point boundary-value problem with assigned boundary values for a system of ordinary differential equations, using a deferred correction technique and a Newton iteration.

2. Specification

SUBROUTINE D02GAF (U, V, N, A, B, TOL, FCN, MNP, X, Y, NP, 1
                  W, LW, IW, LIW, IFAIL)

  INTEGER N, MNP, NP, LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION U(N,2), V(N,2), A, B, TOL, X(MNP), Y
  1 (N,MNP), W(LW)
  EXTERNAL FCN

3. Description

D02GAF solves a two-point boundary-value problem for a system of n differential equations in the interval [a,b]. The system is written in the form

\[ y' = f(x, y_1, y_2, \ldots, y_n), \quad i=1,2,\ldots,n \quad (1) \]

and the derivatives are evaluated by a subroutine FCN supplied by the user. Initially, n boundary values of the variables \( y_i \) must be specified (assigned), some at a and some at b. The user also supplies estimates of the remaining n boundary values and all the boundary values are used in constructing an initial approximation to the solution. This approximate solution is corrected by a finite-difference technique with deferred correction allied with a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1]. The Newton iteration requires a Jacobian matrix \( \frac{\partial f}{\partial y} \) and this is calculated by numerical differentiation using an algorithm described in Curtis et al [2].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference
equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user’s tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If on the other hand the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References


5. Parameters

1: U(N,2) -- DOUBLE PRECISION array Input
   On entry: U(i,1) must be set to the known (assigned) or estimated values of y at a and U(i,2) must be set to the known or estimated values of y at b, for i=1,2,...,n.

2: V(N,2) -- DOUBLE PRECISION array Input
   On entry: V(i,j) must be set to 0.0 if U(i,j) is a known (assigned) value and to 1.0 if U(i,j) is an estimated value, i=1,2,...,n; j=1,2. Constraint: precisely N of the V(i,j) must be set to 0.0, i.e., precisely N of the U(i,j) must be known values, and these must not be all at a or all at b.

3: N -- INTEGER Input

4: A -- DOUBLE PRECISION Input
   On entry: the left-hand boundary point, a.

5: B -- DOUBLE PRECISION Input
   On entry: the right-hand boundary point, b. Constraint: B >
6: TOL -- DOUBLE PRECISION
   Input
   On entry: a positive absolute error tolerance. If
   \[ a = x < x < \ldots < x = b \]
   \[ 1 \quad 2 \quad \text{NP} \]
   is the final mesh, \( z_i(x_j) \) is the \( j \)-th component of the
   approximate solution at \( x_i \), and \( y_j(x_i) \) is the \( j \)-th component
   of the true solution of equation (1) (see Section 3) and the
   boundary conditions, then, except in extreme cases, it is
   expected that
   \[ |z_i(x_j) - y_j(x_i)| \leq TOL, \quad i=1,2,\ldots,\text{NP};j=1,2,\ldots,n \]
   Constraint: TOL > 0.0.

7: FCN -- SUBROUTINE, supplied by the user.
   External Procedure
   FCN must evaluate the functions \( f \) (i.e., the derivatives
   \( y'_i \)) at the general point \( x_i \).
   Its specification is:

   \[
   \text{SUBROUTINE FCN (X, Y, F)}
   \text{DOUBLE PRECISION X, Y(n), F(n)}
   \]
   where \( n \) is the actual value of \( N \) in the call of D02GAF.

   1: X -- DOUBLE PRECISION
      Input
      On entry: the value of the argument \( x \).

   2: Y(*) -- DOUBLE PRECISION array
      Input
      On entry: the value of the argument \( y_i \), for
      \[ i=1,2,\ldots,n \].

   3: F(*) -- DOUBLE PRECISION array
      Output
      On exit: the values of \( f_i \), for \( i=1,2,\ldots,n \).

   FCN must be declared as EXTERNAL in the (sub)program
   from which D02GAF is called. Parameters denoted as
   Input must not be changed by this procedure.

8: MNP -- INTEGER
   Input
   On entry: the maximum permitted number of mesh points.
   Constraint: MNP >= 32.

9: X(MNP) -- DOUBLE PRECISION array
   Input/Output
On entry: if \( NP \geq 4 \) (see NP below), the first \( NP \) elements must define an initial mesh. Otherwise the elements of \( X \) need not be set. Constraint:
\[
A = X(1) < X(2) < \ldots < X(NP) = B \quad \text{for} \quad NP = 4
\]
(3)
On exit: \( X(1), X(2), \ldots, X(NP) \) define the final mesh (with the returned value of \( NP \)) satisfying the relation (3).

10: \( Y(N,MNP) \) -- DOUBLE PRECISION array
Output
On exit: the approximate solution \( z(x) \) satisfying (2), on the final mesh, that is
\[
Y(j,i) = z(x), \quad i = 1, 2, \ldots, NP; j = 1, 2, \ldots, n,
\]
where \( NP \) is the number of points in the final mesh.
The remaining columns of \( Y \) are not used.

11: \( NP \) -- INTEGER
Input/Output
On entry: determines whether a default or user-supplied mesh is used. If \( NP = 0 \), a default value of 4 for \( NP \) and a corresponding equispaced mesh \( X(1), X(2), \ldots, X(NP) \) are used. If \( NP = 4 \), then the user must define an initial mesh using the array \( X \) as described. Constraint: \( NP = 0 \) or \( 4 \leq NP \leq MNP \). On exit: the number of points in the final (returned) mesh.

12: \( W(LW) \) -- DOUBLE PRECISION array
Workspace

13: \( LW \) -- INTEGER
Input
On entry: the length of the array \( W \) as declared in the calling (sub)program. Constraint: \( LW \geq MNP*(3N + 6N + 2) + 4N + 4N \)

14: \( IW(LIW) \) -- INTEGER array
Workspace

15: \( LIW \) -- INTEGER
Input
On entry: the length of the array \( IW \) as declared in the calling (sub)program. Constraint: \( LIW \geq MNP*(2N + 1) + N + 4N + 2 \).

16: \( IFAIL \) -- INTEGER
Input/Output
For this routine, the normal use of \( IFAIL \) is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, \( IFAIL \) must be set to a value with the decimal expansion \( cba \), where each of the decimal digits \( c, b \) and \( a \) must have a value of 0 or 1. 
\( a=0 \) specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages will be printed (see Section 6);

c=0 suppresses warning messages, otherwise warning messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
One or more of the parameters N, TOL, NP, MNP, LW or LIW has been incorrectly set, or $B \leq A$, or the condition (3) on $X$ is not satisfied, or the number of known boundary values (specified by V) is not N.

IFAIL= 2
The Newton iteration has failed to converge. This could be due to there being too few points in the initial mesh or to the initial approximate solution being too inaccurate. If this latter reason is suspected the user should use subroutine D02RAF instead. If the warning 'Jacobian matrix is singular' is printed this could be due to specifying zero estimated boundary values and these should be varied. This warning could also be printed in the unlikely event of the Jacobian matrix being calculated inaccurately. If the user cannot make changes to prevent the warning then subroutine D02RAF should be used.

IFAIL= 3
The Newton iteration has reached round-off level. It could be, however, that the answer returned is satisfactory. This error might occur if too much accuracy is requested.

IFAIL= 4
A finer mesh is required for the accuracy requested; that is MNP is not large enough.

IFAIL= 5
A serious error has occurred in a call to D02GAF. Check all array subscripts and subroutine parameter lists in calls to
D02GAF. Seek expert help.

7. Accuracy

The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (2) except in extreme circumstances. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments

The time taken by the routine depends on the difficulty of the problem, the number of mesh points used (and the number of different meshes used), the number of Newton iterations and the number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

A common cause of convergence problems in the Newton iteration is the user specifying too few points in the initial mesh. Although the routine adds points to the mesh to improve accuracy it is unable to do so until the solution on the initial mesh has been calculated in the Newton iteration.

If the user specifies zero known and estimated boundary values, the routine constructs a zero initial approximation and in many cases the Jacobian is singular when evaluated for this approximation, leading to the breakdown of the Newton iteration.

The user may be unable to provide a sufficiently good choice of initial mesh and estimated boundary values, and hence the Newton iteration may never converge. In this case the continuation facility provided in D02RAF is recommended.

In the case where the user wishes to solve a sequence of similar problems, the final mesh from solving one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the differential equation
\[ y''' = -yy'' - (\beta)(1-y') \]

with boundary conditions
\[ y(0) = y'(0) = 0, \]
\[ y'(10) = 1 \]

for \((\beta) = 0.0\) and \((\beta) = 0.2\) to an accuracy specified by \(TOL = 1.0 \times 10^{-3}\). We solve first the simpler problem with \((\beta) = 0.0\) using an equispaced mesh of 26 points and then we solve the problem with \((\beta) = 0.2\) using the final mesh from the first problem.

Note the call to X04ABF prior to the call to D02GAF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

## 2. Specification

```fortran
SUBROUTINE D02GBF (A, B, N, TOL, FCNF, FCNG, C, D, GAM,
  1 MNP, X, Y, NP, W, LW, IW, LIW, IFAIL)
INTEGER N, MNP, NP, LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION A, B, TOL, C(N,N), D(N,N), GAM(N), X(MNP),
  1 Y(N,MNP), W(LW)
EXTERNAL FCNF, FCNG
```

## 3. Description

D02GBF solves the linear two-point boundary value problem for a system of ordinary differential equations in the interval
[a,b]. The system is written in the form

\[ y' = F(x)y + g(x) \quad (1) \]

and the boundary conditions are written in the form

\[ Cy(a) + Dy(b) = (\gamma) \quad (2) \]

Here \( F(x) \), \( C \) and \( D \) are \( n \) by \( n \) matrices, and \( g(x) \) and \( (\gamma) \) are \( n \)-component vectors. The approximate solution to (1) and (2) is found using a finite-difference method with deferred correction. The algorithm is a specialisation of that used in subroutine D02RAF which solves a nonlinear version of (1) and (2). The nonlinear version of the algorithm is described fully in Pereyra [1].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user’s tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points, then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References


5. Parameters

1: A -- DOUBLE PRECISION 
   Input
   On entry: the left-hand boundary point, \( a \).

2: B -- DOUBLE PRECISION 
   Input
   On entry: the right-hand boundary point, \( b \). Constraint: \( B > A \).
3: N -- INTEGER
   On entry: the number of equations; that is n is the order of
   system (1). Constraint: N >= 2.

4: TOL -- DOUBLE PRECISION
   On entry: a positive absolute error tolerance. If
     a = x < x < ... < x = b
       1   2  ...  NP
is the final mesh, z(x) is the approximate solution from
D02GBF and y(x) is the true solution of equations (1) and
(2) then, except in extreme cases, it is expected that
   ||z-y|| <= TOL
   \text{where}
   ||u|| = \max_{1 \leq i \leq N} \max_{1 \leq j \leq NP} |u(x,j)|.
   Constraint: TOL > 0.0.

5: FCNF -- SUBROUTINE, supplied by the user.
   External Procedure
   FCNF must evaluate the matrix F(x) in (1) at a general point x.

   Its specification is:

   \begin{verbatim}
   SUBROUTINE FCN (X, F)
   DOUBLE PRECISION X, F(n,n)
   where n is the actual value of N in the call of D02GBF.
   \end{verbatim}

     1: X -- DOUBLE PRECISION
        On entry: the value of the independent variable x.

     2: F(n,n) -- DOUBLE PRECISION array
        On exit: the (i,j)th element of the matrix F(x), for
        i,j=1,2,...,n. (See Section 9 for an example.)

   FCN must be declared as EXTERNAL in the (sub)program
   from which D02GBF is called. Parameters denoted as
   Input must not be changed by this procedure.

6: FCNG -- SUBROUTINE, supplied by the user.
   External Procedure
   FCNG must evaluate the vector g(x) in (1) at a general point x.

   Its specification is:

   \begin{verbatim}
   SUBROUTINE FCNG (X, G)
   DOUBLE PRECISION X, G(n)
   where n is the actual value of N in the call of D02GBF.
   \end{verbatim}

     1: X -- DOUBLE PRECISION
        Input
On entry: the value of the independent variable x.

2: G(*) -- DOUBLE PRECISION array Output
On exit: the ith element of the vector g(x), for
i=1,2,...,n. (See Section 9 for an example.)

FCNG must be declared as EXTERNAL in the (sub)program
from which D02GBF is called. Parameters denoted as
Input must not be changed by this procedure.

7: C(N,N) -- DOUBLE PRECISION array Input/Output
8: D(N,N) -- DOUBLE PRECISION array Input/Output
9: GAM(N) -- DOUBLE PRECISION array Input/Output
On entry: the arrays C and D must be set to the matrices C
and D in (2). GAM must be set to the vector (gamma) in (2).
On exit: the rows of C and D and the components of GAM are
re-ordered so that the boundary conditions are in the order:
(i) conditions on y(a) only;
(ii) condition involving y(a) and y(b); and
(iii) conditions on y(b) only.

The routine will be slightly more efficient if the arrays C,
D and GAM are ordered in this way before entry, and in this
event they will be unchanged on exit.

Note that the problems (1) and (2) must be of boundary value
type, that is neither C nor D may be identically zero. Note
also that the rank of the matrix [C,D] must be n for the
problem to be properly posed. Any violation of these
conditions will lead to an error exit.

10: MNP -- INTEGER Input
On entry: the maximum permitted number of mesh points.
Constraint: MNP >= 32.

11: X(MNP) -- DOUBLE PRECISION array Input/Output
On entry: if NP >= 4 (see NP below), the first NP elements
must define an initial mesh. Otherwise the elements of x
need not be set. Constraint:
A=X(1)<X(2)<...<X(NP)=B, for NP>=4.
(4)
On exit: X(1),X(2),...,X(NP) define the final mesh (with the
returned value of NP) satisfying the relation (4).

12: Y(N,MNP) -- DOUBLE PRECISION array Output
On exit: the approximate solution z(x) satisfying (3), on
the final mesh, that is
Y(j,i)=z(x), i=1,2,...,NP; j=1,2,...,n
where \( NP \) is the number of points in the final mesh.

The remaining columns of \( Y \) are not used.

13: \( NP \) -- INTEGER Input/Output
On entry: determines whether a default mesh or user-supplied mesh is used. If \( NP = 0 \), a default value of 4 for \( NP \) and a corresponding equispaced mesh \( X(1), X(2), \ldots, X(NP) \) are used. If \( NP \geq 4 \), then the user must define an initial mesh \( X \) as in (4) above. On exit: the number of points in the final (returned) mesh.

14: \( W(LW) \) -- DOUBLE PRECISION array Workspace

15: \( LW \) -- INTEGER Input
On entry: the length of the array \( W \), Constraint:
\[
LW \geq MNP \times (3N + 5N + 2) + 3N + 5N.
\]

16: \( IW(LIW) \) -- INTEGER array Workspace

17: \( LIW \) -- INTEGER Input
On entry: the length of the array \( IW \), Constraint:
\[
LIW \geq MNP \times (2N + 1) + N.
\]

18: \( IFAIL \) -- INTEGER Input/Output
For this routine, the normal use of \( IFAIL \) is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, \( IFAIL \) must be set to a value with the decimal expansion \( cba \), where each of the decimal digits \( c, b \) and \( a \) must have a value of 0 or 1.
- \( a = 0 \) specifies hard failure, otherwise soft failure;
- \( b = 0 \) suppresses error messages, otherwise error messages will be printed (see Section 6);
- \( c = 0 \) suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), \( IFAIL \) contains 0 on exit.

6. Error Indicators and Warnings
Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL= 1
One or more of the parameters N, TOL, NP, MNP, LW or LIW is incorrectly set, B <= A or the condition (4) on X is not satisfied.

IFAIL= 2
There are three possible reasons for this error exit to be taken:
(i) one of the matrices C or D is identically zero (that is the problem is of initial value and not boundary value type). In this case, IW(1) = 0 on exit;

(ii) a row of C and the corresponding row of D are identically zero (that is the boundary conditions are rank deficient). In this case, on exit IW(1) contains the index of the first such row encountered; and

(iii) more than n of the columns of the n by 2n matrix [C,D]

are identically zero (that is the boundary conditions are rank deficient). In this case, on exit IW(1) contains minus the number of non-identically zero columns.

IFAIL= 3
The routine has failed to find a solution to the specified accuracy. There are a variety of possible reasons including:
(i) the boundary conditions are rank deficient, which may be indicated by the message that the Jacobian is singular. However this is an unlikely explanation for the error exit as all rank deficient boundary conditions should lead instead to error exits with either IFAIL = 2 or IFAIL = 5; see also (iv) below;

(ii) not enough mesh points are permitted in order to attain the required accuracy. This is indicated by NP = MNP on return from a call to D02GBF. This difficulty may be aggravated by a poor initial choice of mesh points;

(iii) the accuracy requested cannot be attained on the computer being used; and

(iv) an unlikely combination of values of F(x) has led to a singular Jacobian. The error should not persist if
more mesh points are allowed.

IFAIL= 4

A serious error has occurred in a call to D02GBF. Check all array subscripts and subroutine parameter lists in calls to D02GBF. Seek expert help.

IFAIL= 5

There are two possible reasons for this error exit which occurs when checking the rank of the boundary conditions by reduction to a row echelon form:

(i) at least one row of the n by 2n matrix \([C, D]\) is a linear combination of the other rows and hence the boundary conditions are rank deficient. The index of the first such row encountered is given by IW(1) on exit; and

(ii) as (i) but the rank deficiency implied by this error exit has only been determined up to a numerical tolerance. Minus the index of the first such row encountered is given by IW(1) on exit.

In case (ii) above there is some doubt as to the rank deficiency of the boundary conditions. However even if the boundary conditions are not rank deficient they are not posed in a suitable form for use with this routine.

For example, if

\[
\begin{pmatrix}
gamma_1 \\ 0 \\ 0 \\ \epsilon \\
\end{pmatrix} = \begin{pmatrix}
gamma_1 \\ 0 \\ 0 \\ \epsilon \\
\end{pmatrix}, \quad D = \begin{pmatrix}
1 & 0 \\ 0 & 0 \\
\end{pmatrix}, \quad (\gamma_1) = (\gamma_1 - \epsilon)
\]

and \((\epsilon)\) is small enough, this error exit is likely to be taken. A better form for the boundary conditions in this case would be

\[
\begin{pmatrix}
gamma_1 \\ 0 \\ 0 \\ \epsilon \\
\end{pmatrix} = \begin{pmatrix}
gamma_1 \\ 0 \\ 0 \\ \epsilon \\
\end{pmatrix}, \quad (\gamma_1) = (\gamma_1 - \epsilon)
\]

7. Accuracy

The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (3) except in extreme circumstances. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments
The time taken by the routine depends on the difficulty of the problem, the number of mesh points (and meshes) used and the number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

In the case where the user wishes to solve a sequence of similar problems, the use of the final mesh from one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the problem (written as a first order system)

\[(\text{epsilon})y'' + y' = 0\]

with boundary conditions

\[y(0) = 0, \quad y(1) = 1\]

for the cases (epsilon)=10 and (epsilon)=10 using the default initial mesh in the first case, and the final mesh of the first case as initial mesh for the second (more difficult) case. We give the solution and the error at each mesh point to illustrate the accuracy of the method given the accuracy request TOL=1.0E-3.

Note the call to X04ABF prior to the call to D02GBF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D02KEF finds a specified eigenvalue of a regular singular second-order Sturm-Liouville system on a finite or infinite range, using a Prüfer transformation and a shooting method. It also reports values of the eigenfunction and its derivatives. Provision is made for discontinuities in the coefficient functions or their derivatives.

2. Specification

SUBROUTINE D02KEF (XPOINT, M, MATCH, COEFFN, BDYVAL, K,  
1 TOL, ELAM, DELAM, HMAX, MAXIT, MAXFUN,  
2 MONIT, REPORT, IFAIL)

INTEGER M, MATCH, K, MAXIT, MAXFUN, IFAIL

DOUBLE PRECISION XPOINT(M), TOL, ELAM, DELAM, HMAX(2,M)

EXTERNAL COEFFN, BDYVAL, MONIT, REPORT

3. Description

D02KEF has essentially the same purpose as D02KDF(*) with minor modifications to enable values of the eigenfunction to be obtained after convergence to the eigenvalue has been achieved.

It first finds a specified eigenvalue (\lambda) of a Sturm-Liouville system defined by a self-adjoint differential equation of the second-order

\[(p(x)y')' + q(x;\lambda)y = 0, \ a < x < b\]

together with the appropriate boundary conditions at the two (finite or infinite) end-points a and b. The functions p and q, which are real-valued, must be defined by a subroutine COEFFN. The boundary conditions must be defined by a subroutine BDYVAL, and, in the case of a singularity at a or b, take the form of an asymptotic formula for the solution near the relevant end-point.

When the final estimate (\lambda) = (\lambda) of the eigenvalue has been found, the routine integrates the differential equation once more with that value of (\lambda), and with initial conditions chosen so that the integral

\[\int_a^b \frac{y(x)}{2ddq} \mid y(x) - \frac{x;\lambda)dx}{\lambda dx} \]

is approximately one. When q(x;\lambda) is of the form
\( (\lambda)w(x) + q(x) \), which is the most common case, \( S \) represents the square of the norm of \( y \) induced by the inner product

\[
\int_a^b \langle f, g \rangle = \int_a^b f(x)g(x)w(x)dx,
\]

with respect to which the eigenfunctions are mutually orthogonal. This normalisation of \( y \) is only approximate, but experience shows that \( S \) generally differs from unity by only one or two per cent.

During this final integration the REPORT routine supplied by the user is called at each integration mesh point \( x \). Sufficient information is returned to permit the user to compute \( y(x) \) and \( y'(x) \) for printing or plotting. For reasons described in Section 8.2, D02KEF passes across to REPORT, not \( y \) and \( y' \), but the Prufer variables \( (\beta) \), \( (\phi) \) and \( (\rho) \) on which the numerical method is based. Their relationship to \( y \) and \( y' \) is given by the equations

\[
p(x)y' = \sqrt{(\beta)} \exp\left(\frac{1}{2} (\rho)\right) \cos\left(\frac{1}{2} (\phi)\right);
\]

\[
y = \frac{1}{\sqrt{(\beta)}} \exp\left(\frac{1}{2} (\rho)\right) \sin\left(\frac{1}{2} (\phi)\right).
\]

For the theoretical basis of the numerical method to be valid, the following conditions should hold on the coefficient functions:

(a) \( p(x) \) must be non-zero and of one sign throughout the interval \((a, b)\); and,

\[
\frac{dq}{d\lambda}
\]

(b) \( \frac{d}{d\lambda} \) must be of one sign throughout \((a, b)\) for all relevant values of \( (\lambda) \), and must not be identically zero as \( x \) varies, for any \( (\lambda) \).

Points of discontinuity in the functions \( p \) and \( q \) or their derivatives are allowed, and should be included as 'break-points' in the array XPOINT.

A good account of the theory of Sturm-Liouville systems, with
some description of Pruefer transformations, is given in Birkhoff and Rota [4], Chapter X. An introduction for the user of Pruefer transformations for the numerical solution of eigenvalue problems arising from physics and chemistry is Bailey [2].

The scaled Pruefer method is fairly recent, and is described in a short note by Pryce [6] and in some detail in the technical report [5].

4. References


5. Parameters

1: XPOINT(M) -- DOUBLE PRECISION array Input
On entry: the points where the boundary conditions computed by BDYVAL are to be imposed, and also any break-points, i.e., XPOINT(1) to XPOINT(m) must contain values x ,...,x
1 2 3 m
such that
x <=x <x <...<x <=x
1 2 3 m-1 m
with the following meanings:
(a) x and x are the left and right end-points, a and b,
1 m
of the domain of definition of the Sturm-Liouville system if these are finite. If either a or b is infinite, the corresponding value x or x may be a
1 m
more-or-less arbitrarily 'large' number of appropriate
(b) $x_2$ and $x_{m-1}$ are the Boundary Matching Points (BMP’s), that is the points at which the left and right boundary conditions computed in BDYVAL are imposed.

If the left-hand end-point is a regular point then the user should set $x = x_0$ (=a), while if it is a singular point the user must set $x > x_1$. Similarly $x_2 = x_0$ (=b) if the right-hand end-point is regular, and $x < x_{m-1}$ if it is singular.

(c) The remaining $m-4$ points $x_3, ..., x_{m-2}$, if any, define ‘break-points’ which divide the interval $[x_2, x_{m-1}]$ into $m-3$ sub-intervals

$$i = [x_1, x_2], ..., i = [x_{m-3}, x_{m-2}], i = [x_{m-2}, x_{m-1}]$$

Numerical integration of the differential equation is stopped and restarted at each break-point. In simple cases no break-points are needed. However if $p(x)$ or $q(x; \lambda)$ are given by different formulae in different parts of the range, then integration is more efficient if the range is broken up by break-points in the appropriate way. Similarly points where any jumps occur in $p(x)$ or $q(x; \lambda)$, or in their derivatives up to the fifth order, should appear as break-points.

Constraint: $X(1) <= X(2) < ... < X(M-1) <= X(M)$.

2: $M$ -- INTEGER Input
On entry: the number of points in the array XPOINT.
Constraint: $M >= 4$.

3: MATCH -- INTEGER Input/Output
On entry: MATCH must be set to the index of the ‘break-point’ to be used as the matching point (see Section 8.3). If MATCH is set to a value outside the range $[2, m-1]$ then a default value is taken, corresponding to the break-point nearest the centre of the interval $[XPOINT(2), XPOINT(m-1)]$. On exit: the index of the break-point actually used as the matching point.

4: COEFFN -- SUBROUTINE, supplied by the user.

External Procedure
COEFFN must compute the values of the coefficient functions $p(x)$ and $q(x;\lambda)$ for given values of $x$ and $\lambda$.
Section 3 states conditions which $p$ and $q$ must satisfy.

Its specification is:

```fortran
SUBROUTINE COEFFN (P, Q, DQDL, X, ELAM, JINT)
DOUBLE PRECISION P, Q, DQDL, X, ELAM
INTEGER JINT

1: P -- DOUBLE PRECISION          Output
   On exit: the value of $p(x)$ for the current value of $x$.

2: Q -- DOUBLE PRECISION          Output
   On exit: the value of $q(x;\lambda)$ for the current
   value of $x$ and the current trial value of $\lambda$.

3: DQDL -- DOUBLE PRECISION       Output
   \(\frac{d}{dx} \frac{p(x;\lambda)}{\lambda}\)
   On exit: the value of \(\frac{d}{d\lambda} \frac{p(x;\lambda)}{\lambda}\) for the
   current value of $x$ and the current trial value of
   $\lambda$. However DQDL is only used in error estimation
   and an approximation (say to within 20%) will suffice.

4: X -- DOUBLE PRECISION          Input
   On entry: the current value of $x$.

5: ELAM -- DOUBLE PRECISION       Input
   On entry: the current trial value of the eigenvalue
   parameter $\lambda$.

6: JINT -- INTEGER                Input
   On entry: the index $j$ of the sub-interval $i$ (see
   specification of XPOINT) in which $x$ lies.
See Section 8.4 and Section 9 for examples.
```

COEFFN must be declared as EXTERNAL in the (sub)program
from which D02KEF is called. Parameters denoted as
Input must not be changed by this procedure.

5: BDYVAL -- SUBROUTINE, supplied by the user.

External Procedure

BDYVAL must define the boundary conditions. For each end-
point, BDYVAL must return (in YL or YR) values of $y(x)$ and
$p(x)y'(x)$ which are consistent with the boundary conditions
at the end-points; only the ratio of the values matters.
Here $x$ is a given point (XL or XR) equal to, or close to,
the end-point.
For a regular end-point (a, say), x=a; and a boundary condition of the form
\[ c_1 y(a) + c_2 y'(a) = 0 \]
can be handled by returning constant values in YL, e.g.
\[ YL(1) = c_1 \quad \text{and} \quad YL(2) = -c_2 p(a). \]

For a singular end-point however, YL(1) and YL(2) will in general be functions of XL and ELAM, and YR(1) and YR(2) functions of XR and ELAM, usually derived analytically from a power-series or asymptotic expansion. Examples are given in Section 8.5 Section 9.

Its specification is:

SUBROUTINE BDYVAL (XL, XR, ELAM, YL, YR)
DOUBLE PRECISION XL, XR, ELAM, YL(3), YR(3)
1: XL -- DOUBLE PRECISION Input
   On entry: if a is a regular end-point of the system (so that a=x_1 =x_2), then XL contains a. If a is a singular point (so that a<=x<x_2), then XL contains a point x_1<x<=x_2.
2: XR -- DOUBLE PRECISION Input
   On entry: if b is a regular end-point of the system (so that x_1=x_2=b), then XR contains b. If b is a singular point (so that x_1<x<=b), then XR contains a point x_1<x<=x_m.
3: ELAM -- DOUBLE PRECISION Input
   On entry: the current trial value of (lambda).
4: YL(3) -- DOUBLE PRECISION array Output
   On exit: YL(1) and YL(2) should contain values of y(x) and p(x)y'(x) respectively (not both zero) which are consistent with the boundary condition at the left-hand end-point, given by x = XL. YL(3) should not be set.
5: YR(3) -- DOUBLE PRECISION array Output
   On exit: YR(1) and YR(2) should contain values of y(x) and p(x)y'(x) respectively (not both zero) which are
consistent with the boundary condition at the right-hand end-point, given by \( x = x_R \). \( Y_R(3) \) should not be set.

BDYVAL must be declared as EXTERNAL in the (sub)program from which D02KEF is called. Parameters denoted as Input must not be changed by this procedure.

6: K -- INTEGER  
On entry: the index \( k \) of the required eigenvalue when the eigenvalues are ordered
\[
\lambda_0 < \lambda_1 < \lambda_2 < \cdots < \lambda_k 
\]
Constraint: \( K \geq 0 \).

7: TOL -- DOUBLE PRECISION  
On entry: the tolerance parameter which determines the accuracy of the computed eigenvalue. The error estimate held in DELAM on exit satisfies the mixed absolute/relative error test
\[
\text{DELAM} \leq \text{TOL} \times \max(1.0, |\lambda\text{ELAM}|) 
\]
where \( \lambda\text{ELAM} \) is the final estimate of the eigenvalue. \( \text{DELAM} \) is usually somewhat smaller than the right-hand side of (*) but not several orders of magnitude smaller. Constraint: \( \text{TOL} > 0.0 \).

8: ELAM -- DOUBLE PRECISION  
On entry: an initial estimate of the eigenvalue \( \lambda \).
On exit: the final computed estimate, whether or not an error occurred.

9: DELAM -- DOUBLE PRECISION  
On entry: an indication of the scale of the problem in the \( \lambda \)-direction. \( \text{DELAM} \) holds the initial 'search step' (positive or negative). Its value is not critical but the first two trial evaluations are made at \( \lambda\text{ELAM} \) and \( \lambda\text{ELAM} + \text{DELAM} \), so the routine will work most efficiently if the eigenvalue lies between these values. A reasonable choice (if a closer bound is not known) is half the distance between adjacent eigenvalues in the neighbourhood of the one sought. In practice, there will often be a problem, similar to the one in hand but with known eigenvalues, which will help one to choose initial values for \( \lambda\text{ELAM} \) and \( \text{DELAM} \).

If \( \text{DELAM} = 0.0 \) on entry, it is given the default value of \( 0.25 \times \max(1.0, |\lambda\text{ELAM}|) \). On exit: with IFAIL = 0, \( \text{DELAM} \) holds an estimate of the absolute error in the computed eigenvalue, that is \( |(\lambda) - \lambda\text{ELAM}| = \text{DELAM} \). (In Section 8.2
we discuss the assumptions under which this is true.) The
ture error is rarely more than twice, or less than a tenth,
of the estimated error.

With IFAIL /= 0, DELAM may hold an estimate of the error, or
its initial value, depending on the value of IFAIL. See
Section 6 for further details.

10: HMAX(2,M) -- DOUBLE PRECISION array
On entry: HMAX(1,j) a maximum step size to be used by the
differential equation code in the jth sub-interval i (as
j described in the specification of parameter XPOINT), for
j=1,2,...,m-3. If it is zero the routine generates a maximum
step size internally.

It is recommended that HMAX(1,j) be set to zero unless the
coefficient functions p and q have features (such as a
narrow peak) within the jth sub-interval that could be 'missed' if a long step were taken. In such a case HMAX(1,j)
should be set to about half the distance over which the
feature should be observed. Too small a value will increase
the computing time for the routine. See Section 8 for
further suggestions.

The rest of the array is used as workspace. On exit: HMAX(
1,m-1) and HMAX(1,m) contain the sensitivity coefficients
(sigma)_l,(sigma)_r, described in Section 8.6. Other entries
contain diagnostic output in case of an error (see Section 6).

11: MAXIT -- INTEGER
On entry: a bound on n, the number of root-finding
r iterations allowed, that is the number of trial values of
(lambda) that are used. If MAXIT <= 0, no such bound is
assumed. (See also under MAXFUN.) Suggested value: MAXIT =
0. On exit: MAXIT will have been decreased by the number of
iterations actually performed, whether or not it was
positive on entry.

12: MAXFUN -- INTEGER
On entry: a bound on n, the number of calls to COEFFN made
f in any one root-finding iteration. If MAXFUN <= 0, no such
bound is assumed. Suggested value: MAXFUN = 0.

MAXFUN and MAXIT may be used to limit the computational cost
of a call to D02KEF, which is roughly proportional to n*n.
External Procedure
MONIT is called by D02KEF at the end of each root-finding iteration and allows the user to monitor the course of the computation by printing out the parameters (see Section 8 for an example).
If no monitoring is required, the dummy subroutine D02KAY may be used. (D02KAY is included in the NAG Foundation Library).

Its specification is:

```
SUBROUTINE MONIT (MAXIT, IFLAG, ELAM, FINFO)
INTEGER MAXIT, IFLAG
DOUBLE PRECISION ELAM, FINFO(15)
```

1: MAXIT -- INTEGER Input
On entry: the current value of the parameter MAXIT of D02KEF; this is decreased by one at each iteration.

2: IFLAG -- INTEGER Input
On entry: IFLAG describes what phase the computation is in, as follows:
IFLAG < 0
an error occurred in the computation of the 'miss-distance' at this iteration;
an error exit from D02KEF with IFAIL = -IFLAG will follow.

IFLAG = 1
the routine is trying to bracket the eigenvalue (lambda).

IFLAG = 2
the routine is converging to the eigenvalue (lambda) (having already bracketed it).

3: ELAM -- DOUBLE PRECISION Input
On entry: the current trial value of (lambda).

4: FINFO(15) -- DOUBLE PRECISION array Input
On entry: information about the behaviour of the shooting method, and diagnostic information in the case of errors. It should not normally be printed in full if no error has occurred (that is, if IFLAG > 0), though
the first few components may be of interest to the user. In case of an error (IFLAG < 0) all the components of FINFO should be printed. The contents of FINFO are as follows:

FINFO(1): the current value of the 'miss-distance' or 'residual' function f((lambda)) on which the shooting method is based. FINFO(1) is set to zero if IFLAG < 0.

FINFO(2): an estimate of the quantity dd(lambda) defined as follows. Consider the perturbation in the miss-distance f((lambda)) that would result if the local error, in the solution of the differential equation, were always positive and equal to its maximum permitted value. Then dd(lambda) is the perturbation in (lambda) that would have the same effect on f((lambda)). Thus, at the zero of f((lambda)),|dd(lambda)| is an approximate bound on the perturbation of the zero (that is the eigenvalue) caused by errors in numerical solution. If dd(lambda) is very large then it is possible that there has been a programming error in COEFFN such that q is independent of (lambda). If this is the case, an error exit with IFAIL = 5 should follow. FINFO(2) is set to zero if IFLAG < 0.

FINFO(3): the number of internal iterations, using the same value of (lambda) and tighter accuracy tolerances, needed to bring the accuracy (that is the value of dd(lambda)) to an acceptable value. Its value should normally be 1.0, and should almost never exceed 2.0.

FINFO(4): the number of calls to COEFFN at this iteration.

FINFO(5): the number of successful steps taken by the internal differential equation solver at this iteration. A step is successful if it is used to advance the integration (cf. COUT(8) in specification of D02PAF(*)).

FINFO(6): the number of unsuccessful steps used by the internal integrator at this iteration (cf. COUT(9) in specification of D02PAF(*)).

FINFO(7): the number of successful steps at the maximum step size taken by the internal integrator at this iteration (cf. COUT(3) in specification of D02PAF(*)).

FINFO(8): is not used.
FINFO(9) to FINFO(15): set to zero, unless IFLAG < 0 in which case they hold the following values describing the point of failure:

FINFO(9): contains the index of the sub-interval where failure occurred, in the range 1 to m-3. In case of an error in BDYVAL, it is set to 0 or m-2 depending on whether the left or right boundary condition caused the error.

FINFO(10): the value f the independent variable x, the point at which error occurred. In case of an error in BDYVAL, it is set to the value of XL or XR as appropriate (see the specification of BDYVAL).

FINFO(11), FINFO(12), FINFO(13): the current values of the Pruefer dependent variables (beta), (phi) and (rho) respectively. These are set to zero in case of an error in BDYVAL.

FINFO(14): the local-error tolerance being used by the internal integrator at the point of failure. This is set to zero in the case of an error in BDYVAL.

FINFO(15): the last integration mesh point. This is set to zero in the case of an error in BDYVAL.

MONIT must be declared as EXTERNAL in the (sub)program from which D02KEF is called. Parameters denoted as Input must not be changed by this procedure.

14: REPORT -- SUBROUTINE, supplied by the user.

This routine provides the means by which the user may compute the eigenfunction y(x) and its derivative at each integration mesh point x. (See Section 8 for an example).

Its specification is:

```fortran
SUBROUTINE REPORT (X, V, JINT)
    INTEGER JINT
    DOUBLE PRECISION X, V(3)
```

1: X -- DOUBLE PRECISION Input
   On entry: the current value of the independent variable x. See Section 8.3 for the order in which values of x are supplied.

2: V(3) -- DOUBLE PRECISION array Input
   On entry: V(1), V(2), V(3) hold the current values of the Pruefer variables (beta), (phi), (rho)
respectively.

3: JINT -- INTEGER  Input
On entry: JINT indicates the sub-interval between break-points in which X lies exactly as for the routine COEFFN, except that at the extreme left end-point (when \( x = \text{XPOINT}(2) \)) JINT is set to 0 and at the extreme right end-point (when \( x = \text{XPOINT}(m-1) \)) JINT is set to \( m-2 \).

REPORT must be declared as EXTERNAL in the (sub)program from which D02KEF is called. Parameters denoted as Input must not be changed by this procedure.

15: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
A parameter error. All parameters (except IFAIL) are left unchanged. The reason for the error is shown by the value of HMAX(2,1) as follows:
HMAX(2,1) = 1: \( M < 4 \);
HMAX(2,1) = 2: \( K < 0 \);
HMAX(2,1) = 3: \( TOL \leq 0.0 \);
HMAX(2,1) = 4: XPOINT(1) to XPOINT(m) are not in ascending order.

HMAX(2,2) gives the position i in XPOINT where this was detected.

IFAIL= 2
At some call to BDYVAL, invalid values were returned, that is, either \( YL(1) = YL(2) = 0.0 \), or \( YR(1) = YR(2) = 0.0 \) (a programming error in BDYVAL). See the last call of MONIT for details.

This error exit will also occur if \( p(x) \) is zero at the point where the boundary condition is imposed. Probably BDYVAL was
called with XL equal to a singular end-point a or with XR
equal to a singular end-point b.

IFAIL = 3
At some point between XL and XR the value of p(x) computed
by COEFFN became zero or changed sign. See the last call of
MONIT for details.

IFAIL = 4
MAXIT > 0 on entry, and after MAXIT iterations the
eigenvalue had not been found to the required accuracy.

IFAIL = 5
The 'bracketing' phase (with parameter IFLAG of MONIT equal
to 1) failed to bracket the eigenvalue within ten
iterations. This is caused by an error in formulating the
problem (for example, q is independent of (lambda)), or by
very poor initial estimates of ELAM, DELAM.

On exit ELAM and ELAM + DELAM give the end-points of the
interval within which no eigenvalue was located by the
routine.

IFAIL = 6
MAXFUN > 0 on entry, and the last iteration was terminated
because more than MAXFUN calls to COEFFN were used. See the
last call of MONIT for details.

IFAIL = 7
To obtain the desired accuracy the local error tolerance was
set so small at the start of some sub-interval that the
differential equation solver could not choose an initial
step size large enough to make significant progress. See the
last call of MONIT for diagnostics.

IFAIL = 8
At some point inside a sub-interval the step size in the
differential equation solver was reduced to a value too
small to make significant progress (for the same reasons as
with IFAIL = 7). This could be due to pathological behaviour
of p(x) and q(x; (lambda)) or to an unreasonable accuracy
requirement or to the current value of (lambda) making the
equations 'stiff'. See the last call of MONIT for details.

IFAIL = 9
TOL is too small for the problem being solved and the
machine precision is being used. The final value of ELAM
should be a very good approximation to the eigenvalue.

IFAIL = 10
C05AZF(*), called by D02KEF, has terminated with the error exit corresponding to a pole of the residual function \( f((\lambda)) \). This error should not occur, but if it does, try solving the problem again with a smaller value for TOL.

IFAIL = 11
A serious error has occurred in an internal call to D02KDY. Check all subroutine calls and array dimensions. Seek expert help.

IFAIL = 12
A serious error has occurred in an internal call to C05AZF(*). Check all subroutine calls and array dimensions. Seek expert help.

HMAX(2,1) holds the failure exit number from the routine where the failure occurred. In the case of a failure in C05AZF(*), HMAX(2,2) holds the value of parameter IND of C05AZF(*).

Note: error exits with IFAIL = 2, 3, 6, 7, 8, 11 are caused by being unable to set up or solve the differential equation at some iteration, and will be immediately preceded by a call of MONIT giving diagnostic information. For other errors, diagnostic information is contained in HMAX(2,j), for j=1,2,...,m, where appropriate.

7. Accuracy

See the discussion in Section 8.2.

8. Further Comments

8.1. Timing

The time taken by the routine depends on the complexity of the coefficient functions, whether they or their derivatives are rapidly changing, the tolerance demanded, and how many iterations are needed to obtain convergence. The amount of work per iteration is roughly doubled when TOL is divided by 16. To make the most economical use of the routine, one should try to obtain good initial values for ELAM and DELAM, and, where appropriate, good asymptotic formulae. The boundary matching points should not be set unnecessarily close to singular points. The extra time needed to compute the eigenfunction is principally the cost of one additional integration once the eigenvalue has been found.

8.2. General Description of the Algorithm
A shooting method, for differential equation problems containing unknown parameters, relies on the construction of a 'miss-distance function', which for given trial values of the parameters measures how far the conditions of the problem are from being met. The problem is then reduced to one of finding the values of the parameters for which the miss-distance function is zero, that is to a root-finding process. Shooting methods differ mainly in how the miss-distance is defined.

This routine defines a miss-distance $f(\lambda)$ based on the rotation around the origin of the point $P(x)=(p(x)y'(x),y(x))$ in the Phase Plane as the solution proceeds from $a$ to $b$. The boundary-conditions define the ray (i.e., two-sided line through the origin) on which $p(x)$ should start, and the ray on which it should finish. The eigenvalue index $k$ defines the total number of half-turns it should make. Numerical solution is actually done by matching point $x=c$. Then $f(\lambda)$ is taken as the angle between the rays to the two resulting points $P(c)$ and $P(c)$. A relative scaling of the $py'$ and $y$ axes, based on the behaviour of the coefficient functions $p$ and $q$, is used to improve the numerical behaviour.

Please see figure in printed Reference Manual

The resulting function $f(\lambda)$ is monotonic over $\lambda$:

- $\lambda < \lambda_0 < \lambda_\infty$, increasing if $\frac{dd}{d\lambda} > 0$ and decreasing $\frac{dd}{d\lambda}

if $\frac{dd}{d\lambda} < 0$, with a unique zero at the desired eigenvalue $\lambda$(lambda). The routine measures $f(\lambda)$ in units of a half-turn. This means that as $\lambda$ increases, $f(\lambda)$ varies by about 1 as each eigenvalue is passed. (This feature implies that the values of $f(\lambda)$ at successive iterations - especially in the early stages of the iterative process - can be used with suitable extrapolation or interpolation to help the choice of initial estimates for eigenvalues near to the one currently being found.)

The routine actually computes a value for $f(\lambda)$ with errors, arising from the local errors of the differential equation code and from the asymptotic formulae provided by the user if singular points are involved. However, the error estimate output in DELAM is usually fairly realistic, in that the actual
error \(|\lambda - \lambda| \) is within an order of magnitude of \( \delta \).

We pass the values of \((\beta), (\phi), (\rho)\) across through \texttt{REPORT} rather than converting them to values of \(y, y'\) inside \texttt{D02KEF}, for the following reasons. First, there may be cases where auxiliary quantities can be more accurately computed from the Pruefer variables than from \(y\) and \(y'\). Second, in singular problems on an infinite interval \(y\) and \(y'\) may underflow towards the end of the range, whereas the Pruefer variables remain well-behaved. Third, with high-order eigenvalues (and therefore highly oscillatory eigenfunctions) the eigenfunction may have a complete oscillation (or more than one oscillation) between two mesh points, so that values of \(y\) and \(y'\) at mesh points give a very poor representation of the curve. The probable behaviour of the Pruefer variables in this case is that \((\beta)\) and \((\rho)\) vary slowly whilst \((\phi)\) increases quickly: for all three Pruefer variables linear interpolation between the values at adjacent mesh points is probably sufficiently accurate to yield acceptable intermediate values of \((\beta), (\phi), (\rho)\) (and hence of \(y, y'\)) for graphical purposes.

Similar considerations apply to the exponentially decaying 'tails Here \((\phi)\) has approximately constant value whilst \((\rho)\) increases rapidly in the direction of integration, though the step length is generally fairly small over such a range.

If the solution is output through \texttt{REPORT} at \(x\)-values which are too widely spaced, the step length can be controlled by choosing \texttt{HMAX} suitably, or, preferably, by reducing \texttt{TOL}. Both these choices will lead to more accurate eigenvalues and eigenfunctions but at some computational cost.

8.3. The Position of the Shooting Matching Point \(c\)

This point is always one of the values \(x_i\) in array \texttt{XPOINT}. It may be specified using the parameter \texttt{MATCH}. The default value is chosen to be the value of that \(x_i, 2 \leq i \leq m-1\), that lies closest to the middle of the interval \([x_2, x_m]\). If there is a tie, the rightmost candidate is chosen. In particular if there are no break-points then \(c=x_2 = (x_m)\) - that is the shooting is from left to right in this case. A break-point may be inserted purely to move \(c\) to an interior point of the interval, even though the form of the equations does not require it. This often speeds up convergence especially with singular problems.

Note that the shooting method used by the code integrates first
from the left-hand end \( x \), then from the right-hand end \( x \), to

\[ 1 \quad r \]

meet at the matching point \( c \) in the middle. This will of course

be reflected in printed or graphical output. The diagram shows a

possible sequence of nine mesh points (\( \tau \)) through (\( \tau \)) in

\[ 1 \quad 9 \]

the order in which they appear, assuming there are just two sub-

intervals (so \( m=5 \)).

Figure 1

Please see figure in printed Reference Manual

Since the shooting method usually fails to match up the two 'legs

\( p(x)y' \) or both, at the matching point \( c \). The code in fact 'shares

large jump does not imply an inaccurate eigenvalue, but implies either

(a) a badly chosen matching point: if \( q(x;\lambda) \) has a '

humped' shape, \( c \) should be chosen near the maximum value of

\( q \), especially if \( q \) is negative at the ends of the interval.

(b) An inherently ill-conditioned problem, typically one where

another eigenvalue is pathologically close to the one being

sought. In this case it is extremely difficult to obtain an

accurate eigenfunction.

In Section 9 below, we find the 11th eigenvalue and corresponding

eigenfunction of the equation

\[ 2y''+((\lambda)-x-2/x)y=0 \quad \text{on} \quad 0<x<\infty \]

the boundary conditions being that \( y \) should remain bounded as \( x \)
tends to 0 and \( x \) tends to \( \infty \). The coding of this problem is
discussed in detail in Section 8.5.

The choice of matching point \( c \) is open. If we choose \( c=30.0 \) as in
the D02KDF(*) example program we find that the exponentially
increasing component of the solution dominates and we get
extremely inaccurate values for the eigenfunction (though the
eigenvalue is determined accurately). The values of the
eigenfunction calculated with \( c=30.0 \) are given schematically in
Figure 2.

Figure 2

Please see figure in printed Reference Manual

If we choose \( c \) as the maximum of the hump in \( q(x;\lambda) \) (see
(a) above) we instead obtain the accurate results given in Figure 3.

Figure 3
Please see figure in printed Reference Manual

8.4. Examples of Coding the COEFFN Routine

Coding COEFFN is straightforward except when break-points are needed. The examples below show:

(a) a simple case,

(b) a case in which discontinuities in the coefficient functions or their derivatives necessitate break-points, and

(c) a case where break-points together with the HMAX parameter are an efficient way to deal with a coefficient function that is well-behaved except over one short interval.

Example A

The modified Bessel equation

\[ x(\sqrt{x}y')' + ((\lambda)x - (\nu))y = 0 \]

Assuming the interval of solution does not contain the origin, dividing through by \(x\), we have \(p(x) = x\),

\[ q(x; (\lambda)) = (\lambda)x - (\nu)/x. \]

The code could be

```fortran
SUBROUTINE COEFFN(P, Q, DQDL, X, ELAM, JINT)
P = X
Q = ELAM*X + NU*NU/X
DQDL = X
RETURN
END
```

where \(NU\) (standing for \((\nu)\)) is a real variable that might be defined in a DATA statement, or might be in user-declared COMMON so that its value could be set in the main program.

Example B

The Schroedinger equation
$y'' + (\lambda + q(x))y = 0$

\[
\begin{align*}
\{ & 2 \\
\{ & x -10 \ (|x|\leq 4),
\end{align*}
\]

where $q(x) = \{ 6/|x| \ (|x|>4),$

over some interval 'approximating to (-infty,infty)', say [-20, 20]. Here we need break-points at ± 4, forming three sub-intervals $i = [-20,-4], i = [-4,4], i = [4,20]$. The code could be

```
1 2 3
```

```
SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
IF (JINT.EQ.2) THEN
   Q = ELAM + X*X - 10.0E0
ELSE
   Q = ELAM + 6.0E0/ABS(X)
ENDIF
P = 1.0E0
DQDL = 1.0E0
RETURN
END
```

The array XPOINT would contain the values $x$, -20.0, -4.0, +4.0, +20.0, $x$ and $m$ would be 6. The choice of appropriate values for $x$ and $x$ depends on the form of the asymptotic formula computed by BDYVAL and the technique is discussed in the next subsection.

Example C

\[
\begin{align*}
\{ & 2 \\
\{ & -100x
\end{align*}
\]

$y'' + (\lambda)(1-2e^{-100x})y = 0$, over $-10 \leq x \leq 10$

Here $q(x; (\lambda))$ is nearly constant over the range except for a sharp inverted spike over approximately $-0.1 \leq x \leq 0.1$. There is a danger that the routine will build up to a large step size and 'step over' the spike without noticing it. By using break-points - say at ± 0.5 - one can restrict the step size near the spike without impairing the efficiency elsewhere.

The code for COEFFN could be
SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
  P = 1.0E0
  DQDL = 1.0E0 - 2.0E0*EXP(-100.0E0*X*X)
  Q = ELAM*DQDL
  RETURN
END

8.5. Examples of Boundary Conditions at Singular Points

Quoting from Bailey [2] page 243: 'Usually... the differential equation has two essentially different types of solution near a singular point, and the boundary condition there merely serves to distinguish one kind from the other. This is the case in all the standard examples of mathematical physics.'

In most cases the behaviour of the ratio \( p(x)y''/y \) near the point is quite different for the two types of solution. Essentially what the user provides through his BDYVAL routine is an approximation to this ratio, valid as \( x \) tends to the singular point (SP).

The user must decide (a) how accurate to make this approximation or asymptotic formula, for example how many terms of a series to use, and (b) where to place the boundary matching point (BMP) at which the numerical solution of the differential equation takes over from the asymptotic formula. Taking the BMP closer to the SP will generally improve the accuracy of the asymptotic formula, but will make the computation more expensive as the Pruefer differential equations generally become progressively more ill-behaved as the SP is approached. The user is strongly recommended to experiment with placing the BMPs. In many singular problems quite crude asymptotic formulae will do. To help the user avoid needlessly accurate formulae, D02KEF outputs two 'sensitivity coefficients' (\( \sigma_l \)) (\( \sigma_r \)) which estimate how much the errors at the BMP's affect the computed eigenvalue. They are described in detail below, see Section 8.6.

Example of coding BDYVAL:

The example below illustrates typical situations:

\[
\begin{align*}
y'' + ((\lambda) - x - ...) y &= 0, \quad 0 < x < \infty
\end{align*}
\]
the boundary conditions being that \( y \) should remain bounded as \( x \) tends to 0 and \( x \) tends to \( \infty \).

At the end \( x=0 \) there is one solution that behaves like \( x \) and another that behaves like \( \frac{1}{x} \). For the first of these solutions \( p(x)y'/y \) is asymptotically \( \frac{2}{x} \) while for the second it is asymptotically \( \frac{-1}{x} \). Thus the desired ratio is specified by setting

\[
YL(1)=x \text{ and } YL(2)=2.0.
\]

At the end \( x=\infty \) the equation behaves like Airy's equation shifted through \((\lambda)\), i.e., like \( y''-ty=0 \) where \( t=x-(\lambda) \), so again there are two types of solution. The solution we require behaves as

\[
\left( \frac{2}{3/2} \right)^{4/3} \exp\left(-\frac{t}{\sqrt{t}}\right)
\]

and the other as

\[
\left( \frac{2}{3/2} \right)^{4/3} \exp\left(+\frac{t}{\sqrt{t}}\right)
\]

once, the desired solution has \( p(x)y'/y\sim\sqrt{t} \) so that we could set

\[
YR(1)=1.0 \text{ and } YR(2)=-\sqrt{x-(\lambda)}.
\]

The complete subroutine might thus be

```fortran
SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)
real XL, XR, ELAM, YL(3), YR(3)
YL(1) = XL
YL(2) = 2.0E0
YR(1) = 1.0E0
YR(2) = -SQRT(XR - ELAM)
RETURN
END
```

Clearly for this problem it is essential that any value given by 
D02KEF to \( XR \) is well to the right of the value of \( ELAM \), so that the user must vary the right-hand BMP with the eigenvalue index \( k \)
k

function $A_i(x)$, so there is no problem in estimating $\lambda$. More accurate asymptotic formulae are easily found - near $x=0$ by
the standard Frobenius method, and near $x=\infty$ by using standard
asymptotics for $A_i(x), A_i'(x)$ (see [1], p. 448). For example, by
the Frobenius method the solution near $x=0$ has the expansion

\[
y(x) = x^2 \left( c_0 + c_1 x + c_2 x^2 + \ldots \right)
\]

with

\[
\begin{array}{cccc}
\lambda & c & -\lambda c \\
n-3 & n-2 & & \\
0 & 1 & 2 & 10 \quad 3 \quad 18 \quad n \quad n(n+3)
\end{array}
\]

This yields

\[
p(x)y' = \frac{2 - (\lambda)x + \ldots}{5},
\]

The sensitivity parameters $(\sigma)_l$ and $(\sigma)_r$ (held in $H(1,m-1)$
and $H(1,m)$ on output) estimate the effect of errors in the
boundary conditions. For sufficiently small errors $(\Delta)y$, $(\Delta)y'$ in $y$ and $y'$ respectively, the relations

\[
(\Delta)(\lambda) \approx (y.(\Delta)y' - y'.(\Delta)y) (\sigma)_l
\]

\[
(\Delta)(\lambda) \approx (y.(\Delta)y' - y'.(\Delta)y) (\sigma)_r
\]

are satisfied where the subscripts $l$, $r$ denote errors committed
at left- and right-hand BMP’s respectively, and $(\Delta)(\lambda)$
denotes the consequent error in the computed eigenvalue.

8.7. Missed Zeros
This is a pitfall to beware of at a singular point. If the BMP is chosen so far from the SP that a zero of the desired eigenfunction lies in between them, then the routine will fail to number of zeros of its eigenfunction, the result will be that:

(a) The wrong eigenvalue will be computed for the given index $k$ - in fact some $(\lambda)$ will be found where $k'>=1$.

(b) The same index $k$ can cause convergence to any of several eigenvalues depending on the initial values of ELAM and DELAM.

It is up to the user to take suitable precautions - for instance by varying the position of the BMP's in the light of his knowledge of the asymptotic behaviour of the eigenfunction at different eigenvalues.

9. Example

To find the 11th eigenvalue and eigenfunction of the example of Section 8.5, using the simple asymptotic formulae for the boundary conditions.

Comparison of the results from this example program with the corresponding results from D02KDF(*) example program shows that similar output is produced from the routine MONIT, followed by the eigenfunction values from REPORT, and then a further line of information from MONIT (corresponding to the integration to find the eigenfunction). Final information is printed within the example program exactly as with D02KDF(*).

Note the discrepancy at the matching point $c(=\sqrt{4}$, the maximum of $q(x;(\lambda))$, in this case) between the solutions obtained by integrations from left and right end-points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
D02 -- Ordinary Differential Equations
D02RAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

D02RAF solves the two-point boundary-value problem with general boundary conditions for a system of ordinary differential equations, using a deferred correction technique and Newton iteration.

2. Specification

```fortran
SUBROUTINE D02RAF (N, MNP, NP, NUMBEG, NUMMIX, TOL, INIT,
                  X, Y, IY, ABT, FCN, G, IJAC, JACOBF,
                  JACOBG, DELEPS, JACEPS, JACGEP, WORK,
                  LWORK, IWORK, LIWORK, IFAIL)

INTEGER N, MNP, NP, NUMBEG, NUMMIX, INIT, IY,
        IJAC, LWORK, IWORK(LIWORK), LIWORK, IFAIL
DOUBLE PRECISION TOL, X(MNP), Y(IY,MNP), ABT(N), DELEPS,
                  WORK(LWORK)
EXTERNAL FCN, G, JACOBF, JACOBG, JACEPS, JACGEP
```

3. Description

D02RAF solves a two-point boundary-value problem for a system of n ordinary differential equations in the interval (a,b) with b>a. The system is written in the form

\[ y_i' = f(x, y_i, y_1, y_2, ..., y_n), \quad i = 1, 2, ..., n \]  \( (1) \)

and the derivatives \( f \) are evaluated by a subroutine FCN supplied by the user. With the differential equations (1) must be given a system of n (nonlinear) boundary conditions

\[ g(y(a), y(b)) = 0, \quad i = 1, 2, ..., n \]  \( i \)  \( (2) \)

where

\[ y(x) = [y_1(x), y_2(x), ..., y_n(x)]^T \]  \( 1 \)  \( 2 \)  \( n \)

The functions \( g \) are evaluated by a subroutine G supplied by the user. The solution is computed using a finite-difference technique with deferred correction allied to a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1].
The user must supply an absolute error tolerance and may also supply an initial mesh for the finite-difference equations and an initial approximate solution (alternatively a default mesh and approximation are used). The approximate solution is corrected using Newton iteration and deferred correction. Then, additional points are added to the mesh and the solution is recomputed with the aim of making the error everywhere less than the user's tolerance and of approximately equidistributing the error on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

The Newton iteration requires Jacobian matrices
\[
\begin{pmatrix}
\frac{df}{dx} & \frac{dg}{dx} & \frac{dg}{dx} \\
\frac{df}{dy} & \frac{dg}{dy} & \frac{dg}{dy} \\
\frac{df}{dy} & \frac{dg}{dy} & \frac{dg}{dy}
\end{pmatrix}
\]
These may be supplied by the user through subroutines JACOB for \( \frac{df}{dx} \)
\( \frac{df}{dy} \) and JACOBG for the others. Alternatively the Jacobians may be calculated by numerical differentiation using the algorithm described in Curtis et al [2].

For problems of the type (1) and (2) for which it is difficult to determine an initial approximation from which the Newton iteration will converge, a continuation facility is provided. The user must set up a family of problems
\[
y' = f(x,y,(\varepsilon)), \quad g(y(a),y(b),(\varepsilon)) = 0, \quad (3)
\]
where \( f = [f_1, f_2, \ldots, f_n] \) etc, and where \( (\varepsilon) \) is a continuation parameter. The choice \( (\varepsilon) = 0 \) must give a problem (3) which is easy to solve and \( (\varepsilon) = 1 \) must define the problem whose solution is actually required. The routine solves a sequence of problems with \( (\varepsilon) \) values
\[
0 = (\varepsilon_1) < (\varepsilon_2) < \ldots < (\varepsilon_p) = 1. \quad (4)
\]
The number $p$ and the values $(\epsilon_i)$ are chosen by the routine
so that each problem can be solved using the solution of its
predecessor as a starting approximation. Jacobians $\frac{\partial f}{\partial x}$
and $\frac{\partial g}{\partial x}$ are required and they may be supplied by the
user via routines JACEPS and JACGEP respectively or may be
computed by numerical differentiation.

4. References

Fortran Program for First Order Nonlinear, Ordinary Boundary
Problems. Codes for Boundary Value Problems in Ordinary
Differential Equations. Lecture Notes in Computer Science.
(ed B Childs, M Scott, J W Daniel, E Denman and P Nelson) 76
Springer-Verlag.

Appl. 13 117--119.

5. Parameters

1: $N$ -- INTEGER Input
On entry: the number of differential equations, $n$.
Constraint: $N > 0$.

2: $MNP$ -- INTEGER Input
On entry: $MNP$ must be set to the maximum permitted number
of points in the finite-difference mesh. If LWORK or LIWORK
(see below) is too small then internally $MNP$ will be
replaced by the maximum permitted by these values. (A
warning message will be output if on entry IFAIL is set to
obtain monitoring information.) Constraint: $MNP \geq 32$.

3: $NP$ -- INTEGER Input/Output
On entry: $NP$ must be set to the number of points to be used
in the initial mesh. Constraint: $4 \leq NP \leq MNP$. On exit:
the number of points in the final mesh.

4: $NUMBEG$ -- INTEGER Input
On entry: the number of left-hand boundary conditions (that
is the number involving $y(a)$ only). Constraint: $0 \leq NUMBEG
< N$. 
5: NUMMIX -- INTEGER
On entry: the number of coupled boundary conditions (that is the number involving both y(a) and y(b)). Constraint: 0 <= NUMMIX <= N - NUMBEG.

6: TOL -- DOUBLE PRECISION
On entry: a positive absolute error tolerance. If a = x_1 < x_2 < ... < x_N = b
is the final mesh, z_j(x_i) is the jth component of the approximate solution at x_i, and y_j(x) is the jth component of the true solution of (1) and (2), then, except in extreme circumstances, it is expected that |z_j(x_i) - y_j(x_i)| <= TOL, i=1,2,...,NP; j=1,2,...,n. (5)
Constraint: TOL > 0.0.

7: INIT -- INTEGER
On entry: indicates whether the user wishes to supply an initial mesh and approximate solution (INIT /= 0) or whether default values are to be used, (INIT = 0).

8: X(MNP) -- DOUBLE PRECISION array
On entry: the user must set X(1) = a and X(NP) = b. If INIT = 0 on entry a default equispaced mesh will be used, otherwise the user must specify a mesh by setting X(i) = x_i for i = 2,3,...,NP-1. Constraints:
X(1) < X(NP), if INIT = 0,
X(1) < X(2) < ... < X(NP), if INIT /= 0.
On exit: X(1),X(2),...,X(NP) define the final mesh (with the returned value of NP) and X(1) = a and X(NP) = b.

9: Y(IY,MNP) -- DOUBLE PRECISION array
On entry: if INIT = 0, then Y need not be set.
If INIT /= 0, then the array Y must contain an initial approximation to the solution such that Y(j,i) contains an approximation to y_j(x_i), i=1,2,...,NP; j=1,2,...,n.
On exit: the approximate solution z_j(x_i) satisfying (5) on the final mesh, that is
Y(j,i) = z_j(x_i), i=1,2,...,NP; j=1,2,...,n,
where NP is the number of points in the final mesh. If an
error has occurred then \( Y \) contains the latest approximation to the solution. The remaining columns of \( Y \) are not used.

10: \( IY \) -- INTEGER
    Input
    On entry:
    the first dimension of the array \( Y \) as declared in the (sub)program from which \texttt{D02RAF} is called.
    Constraint: \( IY \geq N \).

11: \( ABT(N) \) -- DOUBLE PRECISION array
    Output
    On exit: \( ABT(i) \), for \( i=1,2,\ldots,n \), holds the largest estimated error (in magnitude) of the \( i \)th component of the solution over all mesh points.

12: \( FCN \) -- SUBROUTINE, supplied by the user.
    External Procedure
    FCN must evaluate the functions \( f \) (i.e., the derivatives \( y' \)) at a general point \( x \) for a given value of \( \epsilon \), the continuation parameter (see Section 3).

    Its specification is:

    ```
    SUBROUTINE FCN (X, EPS, Y, F, N)
    INTEGER N
    DOUBLE PRECISION X, EPS, Y(N), F(N)
    
    1: \( X \) -- DOUBLE PRECISION
    Input
    On entry: the value of the argument \( x \).

    2: \( EPS \) -- DOUBLE PRECISION
    Input
    On entry: the value of the continuation parameter, \( \epsilon \). This is 1 if continuation is not being used.

    3: \( Y(N) \) -- DOUBLE PRECISION array
    Input
    On entry: the value of the argument \( y_i \), for \( i=1,2,\ldots,n \).

    4: \( F(N) \) -- DOUBLE PRECISION array
    Output
    On exit: the values of \( f_i \), for \( i=1,2,\ldots,n \).

    5: \( N \) -- INTEGER
    Input
    On entry: the number of equations.
    FCN must be declared as \texttt{EXTERNAL} in the (sub)program from which \texttt{D02RAF} is called. Parameters denoted as Input must not be changed by this procedure.
13: G -- SUBROUTINE, supplied by the user.

    External Procedure

G must evaluate the boundary conditions in equation (3) and
place them in the array BC.

Its specification is:

    SUBROUTINE G (EPS, YA, YB, BC, N)
    INTEGER N
    DOUBLE PRECISION EPS, YA(N), YB(N), BC(N)

1: EPS -- DOUBLE PRECISION
    Input
    On entry: the value of the continuation parameter,
    (epsilon). This is 1 if continuation is not being used.

2: YA(N) -- DOUBLE PRECISION array
    Input
    On entry: the value y (a), for i=1,2,...,n.

3: YB(N) -- DOUBLE PRECISION array
    Input
    On entry: the value y (b), for i=1,2,...,n.

4: BC(N) -- DOUBLE PRECISION array
    Output
    On exit: the values g (y(a),y(b),(epsilon)), for
    i=1,2,...,n. These must be ordered as follows:
    (i) first, the conditions involving only y(a) (see
        NUMBEG description above);
    (ii) next, the NUMMIX coupled conditions involving
        both y(a) and y(b) (see NUMMIX description
        above); and,
    (iii) finally, the conditions involving only y(b) (N-
        NUMBEG-NUMMIX).

5: N -- INTEGER
    Input
    On entry: the number of equations, n.

G must be declared as EXTERNAL in the (sub)program from
which D02RAF is called. Parameters denoted as Input
must not be changed by this procedure.

14: IJAC -- INTEGER
    Input
    On entry: indicates whether or not the user is supplying
    Jacobian evaluation routines. If IJAC /= 0 then the user
    must supply routines JACOBF and JACOBG and also, when
    continuation is used, routines JACEPS and JACGEP. If IJAC =
    0 numerical differentiation is used to calculate the
    Jacobian and the routines D02GAZ, D02GAY, D02GAZ and D02GAX
JACOBF must evaluate the Jacobian \( \frac{\partial f}{\partial y_j} \) for \( i,j=1,2,\ldots,n \),

\[
\begin{align*}
\frac{\partial f}{\partial y_j} & \quad \text{for } j=1,2,\ldots,n.
\end{align*}
\]

given \( x \) and \( y \), for \( j=1,2,\ldots,n \).

Its specification is:

```plaintext
SUBROUTINE JACOBF (X, EPS, Y, F, N)
INTEGER N
DOUBLE PRECISION X, EPS, Y(N), F(N,N)

1: X -- DOUBLE PRECISION
On entry: the value of the argument \( x \).

2: EPS -- DOUBLE PRECISION
On entry: the value of the continuation parameter (epsilon). This is 1 if continuation is not being used.

3: Y(N) -- DOUBLE PRECISION array
On entry: the value of the argument \( y_i \), for \( i=1,2,\ldots,n \).

4: F(N,N) -- DOUBLE PRECISION array
   On exit: \( F(i,j) \) must be set to the value of \( \frac{\partial f}{\partial y_j} \), evaluated at the point \((x,y)\), for \( i,j=1,2,\ldots,n \).

5: N -- INTEGER
On entry: the number of equations, \( n \).

JACOBF must be declared as EXTERNAL in the (sub)program from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

16: JACOBF -- SUBROUTINE, supplied by the user.

JACOBFG must evaluate the Jacobians \( \frac{\partial f_a}{\partial y_j} \) and \( \frac{\partial f_b}{\partial y_j} \)

\[
\begin{align*}
\frac{\partial f_a}{\partial y_j} \quad \text{and} \quad \frac{\partial f_b}{\partial y_j} 
\end{align*}
\]

The ordering of the rows of \( AJ \) and \( BJ \) must correspond to the ordering of the boundary conditions described in the specification of subroutine G above.
Its specification is:

SUBROUTINE JACOBG (EPS, YA, YB, AJ, BJ, N)
INTEGER N
DOUBLE PRECISION EPS, YA(N), YB(N), AJ(N,N), BJ
1 (N,N)
1: EPS -- DOUBLE PRECISION Input
On entry: the value of the continuation parameter, (epsilon). This is 1 if continuation is not being used.

2: YA(N) -- DOUBLE PRECISION array Input
On entry: the value y (a), for i=1,2,...,n.

3: YB(N) -- DOUBLE PRECISION array Input
On entry: the value y (b), for i=1,2,...,n.

4: AJ(N,N) -- DOUBLE PRECISION array Output
ddg 
i
On exit: AJ(i,j) must be set to the value -------,
ddy (a) 
j
for i,j=1,2,...,n.

5: BJ(N,N) -- DOUBLE PRECISION array Output
ddg 
i
On exit: BJ(i,j) must be set to the value -------,
ddy (b) 
j
for i,j=1,2,...,n.

6: N -- INTEGER Input
On entry: the number of equations, n.
JACOBG must be declared as EXTERNAL in the (sub)program from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

17: DELEPS -- DOUBLE PRECISION Input/Output
On entry: DELEPS must be given a value which specifies whether continuation is required. If DELEPS <= 0.0 or DELEPS >= 1.0 then it is assumed that continuation is not required. If 0.0 < DELEPS < 1.0 then it is assumed that continuation is required unless DELEPS < square root of machine precision when an error exit is taken. DELEPS is used as the increment (epsilon) -(epsilon) (see (4)) and the choice DELEPS = 0.1
is recommended. On exit: an overestimate of the increment 
\( (\epsilon_p)^{-1} \) (in fact the value of the increment 
\( \epsilon_p^{-1} \) which would have been tried if the restriction \( (\epsilon_p)^{-1} = 1 \) 
had not been imposed). If continuation was not requested 
then \( DELEPS = 0.0 \).

If continuation is not requested then the parameters \( JACEPS \) 
and \( JACGEP \) may be replaced by dummy actual parameters in the 
call to \( D02RAF \). (\( D02GAZ \) and \( D02GAX \) respectively may be used 
as the dummy parameters.)

18: \( JACEPS \) -- SUBROUTINE, supplied by the user.

\textbf{External Procedure}

\texttt{ddf}

\texttt{i}

\texttt{JACEPS} must evaluate the derivative \( \frac{d y}{d \epsilon} \) given \( x \) and 
\( \epsilon \) if continuation is being used.

Its specification is:

\begin{verbatim}
SUBROUTINE JACEPS (X, EPS, Y, F, N)
INTEGER N
DOUBLE PRECISION X, EPS, Y(N), F(N)

1: X -- DOUBLE PRECISION Input
On entry: the value of the argument \( x \).

2: EPS -- DOUBLE PRECISION Input
On entry: the value of the continuation parameter, 
(\( \epsilon \)).

3: Y(N) -- DOUBLE PRECISION array Input
On entry: the solution values \( y_i \) at the point \( x \), for 
\( i = 1,2,...,n \).

4: F(N) -- DOUBLE PRECISION array Output
On exit: \( F(i) \) must contain the value \( \frac{d y}{d \epsilon} \) at 
the point \( (x,y) \), for \( i = 1,2,...,n \).

5: N -- INTEGER Input
On entry: the number of equations, \( n \).
\end{verbatim}

\texttt{JACEPS} must be declared as \texttt{EXTERNAL} in the \texttt{(sub)program}
from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

19: JACGEP -- SUBROUTINE, supplied by the user. 

External Procedure 

ddg

JacGEP must evaluate the derivatives 

\[
\text{dd}(\epsilon)
\]

if continuation is being used.

Its specification is:

```
SUBROUTINE JACGEP (EPS, YA, YB, BCEP, N)

INTEGER N
DOUBLE PRECISION EPS, YA(N), YB(N), BCEP(N)

1: EPS -- DOUBLE PRECISION  
   On entry: the value of the continuation parameter, 
   (epsilon).

2: YA(N) -- DOUBLE PRECISION array  
   On entry: the value of \( y(a) \), for \( i=1,2,\ldots,n \).

3: YB(N) -- DOUBLE PRECISION array  
   On entry: the value of \( y(b) \), for \( i=1,2,\ldots,n \).

4: BCEP(N) -- DOUBLE PRECISION array  
   On exit: BCEP(i) must contain the value of 
   \[ 
   \text{ddg}_i 
   \] 
   \[ 
   \text{dd}(\epsilon) 
   \] 
   for \( i=1,2,\ldots,n \).

5: N -- INTEGER  
   On entry: the number of equations, \( n \).

JACGEP must be declared as EXTERNAL in the (sub)program from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

20: WORK(LWORK) -- DOUBLE PRECISION array  
   Workspace

21: LWORK -- INTEGER  
   On entry: the dimension of the array WORK as declared in the 
   (sub)program from which D02RAF is called.

\[
LWORK \geq MNP*(3N + 6N+2)+4N + 3N
\]
22: IWORK(LIWORK) -- INTEGER array

Workspace

23: LIWORK -- INTEGER

Input

On entry:
the dimension of the array IWORK as declared in the
(sub)program from which D02RAF is called.

Constraints:
LIWORK>=(MNP*(2*N+1)+N, if IJAC /= 0,
2
LIWORK>=(MNP*(2*N+1)+N +4*N+2, if IJAC = 0.

24: IFAIL -- INTEGER

Input/Output

For this routine, the normal use of IFAIL is extended to
c Control the printing of error and warning messages as well
as specifying hard or soft failure (see the Essential
Introduction).

Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a
must have a value of 0 or 1.

a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages
will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning
messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e.,
hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL
contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
One or more of the parameters N, MNP, NP, NUMBEG, NUMMIX,
TOL, DELEPS, LWORK or LIWORK has been incorrectly set, or X
(1) >= X(NP) or the mesh points X(i) are not in strictly
ascending order.

IFAIL= 2
A finer mesh is required for the accuracy requested; that is MNP is not large enough. This error exit normally occurs when the problem being solved is difficult (for example, there is a boundary layer) and high accuracy is requested. A poor initial choice of mesh points will make this error exit more likely.

**IFAIL** = 3
The Newton iteration has failed to converge. There are several possible causes for this error:

(i) faulty coding in one of the Jacobian calculation routines;

(ii) if IJAC = 0 then inaccurate Jacobians may have been calculated numerically (this is a very unlikely cause); or,

(iii) a poor initial mesh or initial approximate solution has been selected either by the user or by default or there are not enough points in the initial mesh. Possibly, the user should try the continuation facility.

**IFAIL** = 4
The Newton iteration has reached round-off error level. It could be however that the answer returned is satisfactory. The error is likely to occur if too high an accuracy is requested.

**IFAIL** = 5
The Jacobian calculated by JACOBG (or the equivalent matrix calculated by numerical differentiation) is singular. This may occur due to faulty coding of JACOBG or, in some circumstances, to a zero initial choice of approximate solution (such as is chosen when INIT = 0).

**IFAIL** = 6
There is no dependence on (epsilon) when continuation is being used. This can be due to faulty coding of JACEPS or JACGEP or, in some circumstances, to a zero initial choice of approximate solution (such as is chosen when INIT = 0).

**IFAIL** = 7
DELEPS is required to be less than machine precision for continuation to proceed. It is likely that either the problem (3) has no solution for some value near the current value of (epsilon) (see the advisory print out from D02RAF) or that the problem is so difficult that even with continuation it is unlikely to be solved using this routine. If the latter cause is suspected then using more mesh points
initially may help.

IFAIL = 8
Indicates that a serious error has occurred in a call to D02RAF. Check all array subscripts and subroutine parameter lists in calls to D02RAF. Seek expert help.

IFAIL = 9
Indicates that a serious error has occurred in a call to D02RAR. Check all array subscripts and subroutine parameter lists in calls to D02RAF. Seek expert help.

7. Accuracy
The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (5) except in extreme circumstances. The final error estimate over the whole mesh for each component is given in the array ABT. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments
There are too many factors present to quantify the timing. The time taken by the routine is negligible only on very simple problems.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation.

In the case where the user wishes to solve a sequence of similar problems, the use of the final mesh and solution from one case as the initial mesh is strongly recommended for the next.

9. Example
We solve the differential equation
\[ y'''' = -yy''' - 2(\text{epsilon})(1-y') \]
with (epsilon)=1 and boundary conditions
\[ y(0) = y'(0) = 0, \quad y'(10) = 1 \]

to an accuracy specified by TOL=1.0E-4. The continuation facility is used with the continuation parameter (epsilon) introduced as in the differential equation above and with DELEPS = 0.1.
initially. (The continuation facility is not needed for this problem and is used here for illustration.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

NagOrdinaryDifferentialEquationsPackage (NAGD02)

Exports:

\begin{verbatim}
| package NAGD02 NagOrdinaryDifferentialEquationsPackage |
\end{verbatim}

---

)abbrev package NAGD02 NagOrdinaryDifferentialEquationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Mon Jun 20 17:56:33 1994
++ Description:
++ This package uses the NAG Library to calculate the numerical solution of
++ ordinary differential equations. There are two main types of problem,
++ those in which all boundary conditions are specified at one point
++ (initial-value problems), and those in which the boundary
++ conditions are distributed between two or more points (boundary-
++ value problems and eigenvalue problems). Routines are available
++ for initial-value problems, two-point boundary-value problems and
++ Sturm-Liouville eigenvalue problems.

NagOrdinaryDifferentialEquationsPackage():Exports == Implementation where
S ==> Symbol
Class 1: FortranOutputStackPackage

Actions 1:

Action 2: d02bbf (DoubleFloat, Integer, Integer, Integer, DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, Union(fn: FileName, fp: Asp7(FCN)), Union(fn: FileName, fp: Asp8(OUTPUT))) -> Result

++ d02bbf(xend, m, n, irelab, x, y, tol, ifail, fcn, output)
++ integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method, and returns the solution at points specified by the user.
++ See \downlink{Manual Page}{manpageXXd02bbf}.

Action 3: d02bhf (DoubleFloat, Integer, Integer, DoubleFloat, DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, Union(fn: FileName, fp: Asp9(G)), Union(fn: FileName, fp: Asp7(FCN))) -> Result

++ d02bhf(xend, n, irelab, hmax, x, y, tol, ifail, g, fcn)
++ integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method until a user-specified function of the solution is zero.
++ See \downlink{Manual Page}{manpageXXd02bhf}.

Action 4: d02cjf (DoubleFloat, Integer, Integer, DoubleFloat, String, DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, Union(fn: FileName, fp: Asp9(G)), Union(fn: FileName, fp: Asp7(FCN)), Union(fn: FileName, fp: Asp8(OUTPUT))) -> Result

++ d02cjf(xend, m, n, tol, relabs, x, y, ifail, g, fcn, output)
++ integrates a system of first-order ordinary differential equations over a range with suitable initial conditions, using a variable-order, variable-step Adams method until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.
++ See \downlink{Manual Page}{manpageXXd02cjf}.

Action 5: d02ejf (DoubleFloat, Integer, Integer, String, Integer, DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, Integer, Integer, Integer, Matrix DoubleFloat, DoubleFloat, DoubleFloat, Integer, Integer, Matrix DoubleFloat, Integer, Integer, Union(fn: FileName, fp: Asp9(G)), Union(fn: FileName, fp: Asp7(FCN)), Union(fn: FileName, fp: Asp8(OUTPUT))) -> Result

++ d02ejf(xend, m, n, relabs, iw, x, y, tol, ifail, g, fcn, pederv, output)
++ integrates a stiff system of first-order ordinary differential equations over an interval with suitable initial conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.
++ See \downlink{Manual Page}{manpageXXd02ejf}.

Action 6: d02gaf (Matrix DoubleFloat, Matrix DoubleFloat, Integer, DoubleFloat, DoubleFloat, Integer, Integer, Integer, Matrix DoubleFloat, Integer, Integer, Matrix DoubleFloat, Integer, Integer, Matrix DoubleFloat, Integer, Integer, Union(fn: FileName, fp: Asp8(OUTPUT))) -> Result

++ d02gaf(xend, m, n, irelab, x, y, tol, ifail, g, fcn, pederv, output)
++ integrates a stiff system of differential equations over an interval with suitable initial conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.
CHAPTER 15. CHAPTER N

Integer,Integer,Union(fn:FileName,fp:Asp7(FCN))) -> Result
++ d02gaf(u,v,n,a,b,tol,mnp,lw,liw,x,np,ifail,fcn)
++ solves the two-point boundary-value problem with assigned
++ boundary values for a system of ordinary differential equations,
++ using a deferred correction technique and a Newton iteration.
++ See \downlink{Manual Page}{manpageXXd02gaf}.

d02gbf : (DoubleFloat,DoubleFloat,Integer,DoubleFloat,_
Integer,Integer,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer,_
Union(fn:FileName,fp:Asp77(FCNF)),_
Union(fn:FileName,fp:Asp78(FCNG))) -> Result
++ d02gbf(a,b,n,tol,mnp,lw,liw,c,d,gam,x,np,ifail,fcnf,fcng)
++ solves a general linear two-point boundary value problem
++ for a system of ordinary differential equations using a deferred
++ correction technique.
++ See \downlink{Manual Page}{manpageXXd02gbf}.

d02kef : (Matrix DoubleFloat,Integer,Integer,DoubleFloat,_
Integer,Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,Integer,_
Integer,Union(fn:FileName,fp:Asp10(COEFFN)),_
Union(fn:FileName,fp:Asp80(BDYVAL))) -> Result
++ d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,
++ hmax,maxit,ifail,coeffn,bdyval)
++ finds a specified eigenvalue of a regular singular second-
++ order Sturm-Liouville system on a finite or infinite range, using
++ a Pruefer transformation and a shooting method. It also reports
++ values of the eigenfunction and its derivatives. Provision is
++ made for discontinuities in the coefficient functions or their
++ derivatives.
++ See \downlink{Manual Page}{manpageXXd02kef}.
++ ASP domains Asp12 and Asp33 are used to supply default
++ subroutines for the MONIT and REPORT arguments via their
++ \axiomOp{outputAsFortran} operation.

d02kef : (Matrix DoubleFloat,Integer,Integer,DoubleFloat,_
Integer,Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,Integer,_
Integer,Union(fn:FileName,fp:Asp10(COEFFN)),_
Union(fn:FileName,fp:Asp80(BDYVAL)),FileName,FileName) -> Result
++ d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,
++ hmax,maxit,ifail,coeffn,bdyval,monit,report)
++ finds a specified eigenvalue of a regular singular second-
++ order Sturm-Liouville system on a finite or infinite range, using
++ a Pruefer transformation and a shooting method. It also reports
++ values of the eigenfunction and its derivatives. Provision is
++ made for discontinuities in the coefficient functions or their
++ derivatives.
++ See \downlink{Manual Page}{manpageXXd02kef}.
++ Files \spad{monit} and \spad{report} will be used to define
++ the subroutines for the
++ MONIT and REPORT arguments.
++ See \downlink{Manual Page}{manpageXXd02gbf}.

d02raf : (Integer,Integer,Integer,Integer,_

D02RAF

D02RAF (for the Fortran implementation) solves the two-point boundary-value problem with general boundary conditions for a system of ordinary differential equations, using a deferred correction technique and Newton iteration. See Manual Page for details.

Implementation

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp7(FCN))
import Union(fn:FileName,fp:Asp8(OUTPUT))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(Matrix DoubleFloat)

d02bbf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_
  irelabArg:Integer,xArg:DoubleFloat,yArg:Matrix DoubleFloat,_
  tolArg:DoubleFloat,ifailArg:Integer,_
  fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_
  outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==

  pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
  if fcnArg case fn
    then outputAsFortran(fcnArg.fn)
    else outputAsFortran(fcnArg.fp)
  popFortranOutputStack()$FOP

  pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
  if outputArg case fn
    then outputAsFortran(outputArg.fn)
    else outputAsFortran(outputArg.fp)
  popFortranOutputStack()$FOP

  [(invokeNagman([fcnFilename, outputFilename]$Lisp,_
    "d02bbf",_
    ["result":S,"w":S,"fcn":S,"output":S]$Lisp,_}]}
CHAPTER 15. CHAPTER N

,"fcn"::S,"output"::S]$Lisp_,
[[xendArg::Any,mArg::Any,nArg::Any,irelabArg::Any,ifailArg::Any,_,
tolArg::Any,ifailArg::Any,xArg::Any,_,
yArg::Any,ifailArg::Any,yArg::Any ]$_
@List Any]$Lisp)₴Lisp)₴Lisp
pretend List (Record(key:Symbol,entry:Any))$Result

d02bhf(xendArg:DoubleFloat,mArg::Integer,irelabArg:Integer,_
hmaxArg:DoubleFloat,xArg:DoubleFloat,yArg:Matrix DoubleFloat,_
tolArg:DoubleFloat,ifailArg:Integer,_
gArg:Union(fn:FileName,fp:Asp9(G)),_  
fcnArg:Union(fn:FileName,fp:Asp7(FCN))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
[[invokeNagman([gFilename,fcnFilename]$Lisp,_
  "d02bhf" _,
  ["w"::S,"g"::S,"fcn"::S]$Lisp_,$Lisp_,
  [[xendArg::Any,mArg::Any,irelabArg::Any,ifailArg::Any,_,
  xArg::Any,tolArg::Any,ifailArg::Any,yArg::Any ]$_
@List Any]$Lisp)$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

d02cjf(xendArg:DoubleFloat,mArg::Integer,nArg::Integer,_
tolArg::DoubleFloat,irelabArg:Integer,xArg:DoubleFloat,_
yArg:Matrix DoubleFloat,ifailArg:Integer,_
gArg:Union(fn:FileName,fp:Asp9(G)),_  
fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_  
outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP  
if fcnArg case fn  
then outputAsFortran(fcnArg.fn)  
else outputAsFortran(fcnArg.fp)  
popFortranOutputStack()$FOP  
pushFortranOutputStack(outputFilename := aspFilename "output")$FOP  
if outputArg case fn  
then outputAsFortran(outputArg.fn)  
else outputAsFortran(outputArg.fp)  
popFortranOutputStack()$FOP

\[
\text{invokeNagman([gFilename, fcnFilename, outputFilename]$Lisp,}_{  \text{"d02cjf"}_,  
["xend":S,"m":S,"n":S,"tol":S,"relabs":S,  
,"x":S,"ifail":S,"g":S,"fcn":S,"output":S,  
,"result":S,"y":S,"w":S]$Lisp,  
["result":S,"w":S,"g":S,"fcn":S,"output":S]$Lisp,  
["double":S,"xend":S,"tol":S,["result":S,"m":S,"n":S]$Lisp,  
,"x":S,["y":S,"n":S]$Lisp,["w":S,["":S,["":S,  
["":S,21$Lisp,"n":S]$Lisp,28$Lisp]$Lisp]$Lisp,"g":S,  
,"fcn":S,"output":S]$Lisp,  
,[["integer":S,"m":S,"n":S,"ifail":S]$Lisp,  
,["character":S,"relabs":S]$Lisp]$Lisp,  
["result":S,"x":S,"y":S,"ifail":S]$Lisp,  
,[[xendArg::Any,mArg::Any,nArg::Any,tolArg::Any,relabsArg::Any,_  
xArg::Any,ifailArg::Any,yArg::Any ]])_  
@List Any]$Lisp)\]

\[
d02ejf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_  
relabsArg:String,iwArg:Integer,xArg:DoubleFloat,_  
yArg:Matrix DoubleFloat,tolArg:DoubleFloat,ifailArg:Integer,_  
gArg:Union(fn:FileName,fp:Asp9(G)),_  
fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_  
pedervArg:Union(fn:FileName,fp:Asp31(PEDERV)),_  
outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==  
pushFortranOutputStack(gFilename := aspFilename "g")$FOP  
if gArg case fn  
then outputAsFortran(gArg.fn)  
else outputAsFortran(gArg.fp)  
popFortranOutputStack()$FOP  
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP  
if fcnArg case fn  
then outputAsFortran(fcnArg.fn)  
else outputAsFortran(fcnArg.fp)  
popFortranOutputStack()$FOP  
pushFortranOutputStack(pedervFilename := aspFilename "pederv")$FOP  
if pedervArg case fn  
then outputAsFortran(pedervArg.fn)  
else outputAsFortran(pedervArg.fp)  
popFortranOutputStack()$FOP
CHAPTER 15. CHAPTER N

pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
if outputArg case fn
  then outputAsFortran(outputArg.fn)
  else outputAsFortran(outputArg.fp)
popFortranOutputStack()$FOP
pretend List (Record(key:Symbol,entry:Any))$Result
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
d02gbf(\(aArg::\text{DoubleFloat}, bArg::\text{DoubleFloat}, nArg::\text{Integer}, \text{tolArg::DoubleFloat}, \text{mnpArg::Integer}, \text{lwArg::Integer}, \text{liwArg::Integer}, \text{npArg::Integer}, \text{ifailArg::Integer}, uArg::\text{Any}, \text{vArg::\text{Any}}\))_ 
\text{pretend List (Record(key:Symbol, entry:Any))}$Result

\text{d02kef(\(xpointArg::\text{Matrix DoubleFloat}, mArg::\text{Integer}, kArg::\text{Integer}, \text{tolArg::DoubleFloat}, \text{maxfunArg::Integer}, \text{matchArg::Integer}, \text{elamArg::DoubleFloat}, \text{delamArg::DoubleFloat}, \text{hmaxArg::Matrix DoubleFloat}, \text{maxitArg::Integer}, \text{ifailArg::Integer}, \text{coeffnArg::Union(fn:FileName, fp:Asp10(COEFFN))}, \text{bdyvalArg::Union(fn:FileName, fp:Asp80(BDYVAL))})\)}: Result == 
\text{pretend List (Record(key:Symbol, entry:Any))}$Result
if coeffnArg case fn
    then outputAsFortran(coeffnArg.fn)
    else outputAsFortran(coeffnArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(bdyvalFilename := aspFilename "bdyval")$FOP
if bdyvalArg case fn
    then outputAsFortran(bdyvalArg.fn)
    else outputAsFortran(bdyvalArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran()$Asp12(MONIT)
popFortranOutputStack()$FOP
pushFortranOutputStack(reportFilename := aspFilename "report")$FOP
outputAsFortran()$Asp33(REPORT)
popFortranOutputStack()$FOP
[(invokeNagman([coeffnFilename,bdyvalFilename,monitFilename, _
                reportFilename]$Lisp, 
                "d02kef", 
                ["m":S,"k":S,"tol":S,"maxfun":S,"match":S_,
                ["double":S,["xpoint":S,"m":S]$Lisp,"tol":S_,
                "elam":S,"delam":S,["hmax":S,"maxfun":S,"match":S_,
                "maxit":S,"ifail":S]$Lisp_,
                "ifail":S]$Lisp_,
                [(mArg::Any,kArg::Any,tolArg::Any,maxfunArg::Any,matchArg::Any, _
                elamArg::Any,delamArg::Any,maxitArg::Any,ifailArg::Any, _
                xpointArg::Any,hmaxArg::Any )]_ 
                @List Any]$Lisp)_$Lisp_)
prettend List (Record(key:Symbol,entry:Any))$Result

\texttt{d02kef(xpointArg:Matrix DoubleFloat,mArg:Integer,kArg:Integer, _
    tolArg:DoubleFloat,maxfunArg:Integer,matchArg:Integer, _
    elamArg:DoubleFloat,delamArg:DoubleFloat,hmaxArg:Matrix DoubleFloat, _
    maxitArg:Integer,ifailArg:Integer, _
    coeffnArg:Union(fn:FileName,fp:Asp10(COEFFN)), _
    bdyvalArg:Union(fn:FileName,fp:Asp80(BDYVAL)), _
    monitArg:FileName,reportArg:FileName): Result ==}
pushFortranOutputStack(coeffnFilename := aspFilename "coeffn")$FOP
if coeffnArg case fn
    then outputAsFortran(coeffnArg.fn)
    else outputAsFortran(coeffnArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(bdyvalFilename := aspFilename "bdyval")$FOP
if bdyvalArg case fn
then outputAsFortran(bdyvalArg.fn)
else outputAsFortran(bdyvalArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran(monitArg)
popFortranOutputStack()$FOP
pushFortranOutputStack(reportFilename := aspFilename "report")$FOP
outputAsFortran(reportArg)
popFortranOutputStack()$FOP
[(invokeNagman([coeffnFilename, bdyvalFilename, monitFilename,_
reportFilename]$Lisp,_
"d02kef",_
["m":S,"k":S,"tol":S,"maxfun":S,"match":S,_
["double":S,["xpoint":S,"m":S]$Lisp,"tol":S_,
"elam":S,"delam":S,["hmax":S,2$Lisp,"m":S]$Lisp,_
"coeffn":S,"bdyval":S,"monit":S,"report":S]$Lisp_ ,
["integer":S,"m":S,"k":S,"maxfun":S,"match":S_ ,
"maxit":S,"ifail":S]$Lisp_ ]$Lisp_ _
"d02raf",_
"ifail":S]$Lisp_ ,
[(mArg::Any,kArg::Any,tolArg::Any,maxfunArg::Any_,
matchArg::Any,elamArg::Any,delamArg::Any,maxitArg::Any_,
ifailArg::Any,xpointArg::Any,hamxArg::Any ))_.
@List Any]$Lisp)$Lisp_ $Result
pretend List (Record(key:Symbol,entry:Any))$Result

d02raf(nArg:Integer,mnpArg:Integer,numbegArg:Integer,_
nummixArg:Integer,tolArg:DoubleFloat,initArg:Integer,_
iyArg:Integer,ijacArg:Integer,lworkArg:Integer,_
liworkArg:Integer,npjArg:Integer,xArg:Matrix DoubleFloat,_
yArg:Matrix DoubleFloat,delepsArg:DoubleFloat,ifailArg:Integer,_
fcnArg:Union(fn:FileName,fp:Asp41(FCN,JACOBF,JACEPS)),_
gArg:Union(fn:FileName,fp:Asp42(G,JACUBG,JACGEP))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
then outputAsFortran(fcnArg.fn)
else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
if gArg case fn
then outputAsFortran(gArg.fn)
else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fcnFilename,gFilename]$Lisp,_
"d02raf",_
This package uses the NAG Library to solve partial differential equations.

1. Scope of the Chapter

This chapter is concerned with the solution of partial differential equations.

2. Background to the Problems

The definition of a partial differential equation problem includes not only the equation itself but also the domain of interest and appropriate subsidiary conditions. Indeed, partial differential equations are usually classified as elliptic, hyperbolic or parabolic according to the form of the equation and the form of the subsidiary conditions which must be assigned to produce a well-posed problem. Ultimately it is hoped that this chapter will contain routines for the solution of equations of each of these types together with automatic mesh generation routines and other utility routines particular to the solution of partial differential equations. The routines in this chapter will often call upon routines from the Linear Algebra Chapter F04 -- Simultaneous Linear Equations.

The classification of partial differential equations is easily described in the case of linear equations of the second order in two independent variables, i.e., equations of the form

\[ au + 2bu + cu + du + eu + fu + g = 0, \tag{1} \]

where \( a, b, c, d, e, f \) and \( g \) are functions of \( x \) and \( y \) only. Equation (1) is called elliptic, hyperbolic or parabolic according as \( ac - b^2 \) is positive, negative or zero. Useful definitions of the concepts of elliptic, hyperbolic and parabolic character can also be given for differential equations in more
than two independent variables, for systems and for nonlinear
differential equations.

For elliptic equations, of which Laplace's equation

\[ u + u = 0 \]  
\[ \nabla^2 u = 0 \]  

is the simplest example of second order, the subsidiary
conditions take the form of boundary conditions, i.e., conditions
which provide information about the solution at all points of a
closed boundary. For example, if equation (2) holds in a plane
domain \( D \) bounded by a contour \( C \), a solution \( u \) may be sought
subject to the condition

\[ u = f \quad \text{on} \quad C, \]  

where \( f \) is a given function. The condition (3) is known as a
Dirichlet boundary condition. Equally common is the Neumann
boundary condition

\[ u' = g \quad \text{on} \quad C, \]  

which is one form of a more general condition

\[ u' + fu = g \quad \text{on} \quad C, \]  

where \( u' \) denotes the derivative of \( u \) normal to the contour \( C \) and
\( f \) and \( g \) are given functions. Provided that \( f \) and \( g \) satisfy
certain restrictions, condition (5) yields a well-posed boundary
value problem for Laplace's equation. In the case of the Neumann
problem, one further piece of information, e.g. the value of \( u \) at
a particular point, is necessary for uniqueness of the solution.
Boundary conditions similar to the above are applicable to more
general second order elliptic equations, whilst two such
conditions are required for equations of fourth order.

For hyperbolic equations, the wave equation

\[ u - u = 0 \]  
\[ \frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0 \]  

is the simplest example of second order. It is equivalent to a
first order system

\[ u - v = 0, \quad v - u = 0. \]  
\[ \frac{\partial u}{\partial t} - \frac{\partial v}{\partial x} = 0, \quad \frac{\partial v}{\partial t} - \frac{\partial u}{\partial x} = 0. \]  

The subsidiary conditions may take the form of initial
conditions, i.e., conditions which provide information about the
solution at points on a suitable open boundary. For example, if equation (6) is satisfied for $t>0$, a solution $u$ may be sought such that

$$ u(x,0) = f(x), \quad u(x,0) = g(x), $$

where $f$ and $g$ are given functions. This is an example of an initial value problem, sometimes known as Cauchy’s problem.

For parabolic equations, of which the heat conduction equation

$$ u_t - u_{xx} = 0 $$

is the simplest example, the subsidiary conditions always include some of initial type and may also include some of boundary type. For example, if equation (9) is satisfied for $t>0$ and $0<x<1$, a solution $u$ may be sought such that

$$ u(x,0) = f(x), \quad 0<x<1, $$

and

$$ u(0,t) = 0, \quad u(1,t) = 1, \quad t>0. $$

This is an example of a mixed initial/boundary value problem.

For all types of partial differential equations, finite difference methods (Mitchell and Griffiths [5]) and finite element methods (Wait and Mitchell [9]) are the most common means of solution and such methods obviously feature prominently either in this chapter or in the companion NAG Finite Element Library. Many of the utility routines in this chapter are concerned with the solution of the large sparse systems of equations which arise from the finite difference and finite element methods.

Alternative methods of solution are often suitable for special classes of problems. For example, the method of characteristics is the most common for hyperbolic equations involving time and one space dimension (Smith [7]). The method of lines (see Mikhlin and Smolitsky [4]) may be used to reduce a parabolic equation to a (stiff) system of ordinary differential equations, which may be solved by means of routines from Chapter D02 -- Ordinary Differential Equations. Similarly, integral equation or boundary element methods (Jaswon and Symm [3]) are frequently used for elliptic equations. Typically, in the latter case, the solution of a boundary value problem is represented in terms of certain boundary functions by an integral expression which satisfies the differential equation throughout the relevant domain. The
boundary functions are obtained by applying the given boundary conditions to this representation. Implementation of this method necessitates discretisation of only the boundary of the domain, the dimensionality of the problem thus being effectively reduced by one. The boundary conditions yield a full system of simultaneous equations, as opposed to the sparse systems yielded by the finite difference and finite element methods, but the full system is usually of much lower order. Solution of this system yields the boundary functions, from which the solution of the problem may be obtained, by quadrature, as and where required.

2.1. References


3. Recommendations on Choice and Use of Routines

The choice of routine will depend first of all upon the type of partial differential equation to be solved. At present no special allowances are made for problems with boundary singularities such as may arise at corners of domains or at points where boundary
conditions change. For such problems results should be treated with caution.

Users may wish to construct their own partial differential equation solution software for problems not solvable by the routines described in Sections 3.1 to 3.4 below. In such cases users can employ appropriate routines from the Linear Algebra Chapters to solve the resulting linear systems; see Section 3.5 for further details.

3.1. Elliptic Equations

The routine D03EDF solves a system of seven-point difference equations in a rectangular grid (in two dimensions), using the multigrid iterative method. The equations are supplied by the user, and the seven-point form allows cross-derivative terms to be represented (see Mitchell and Griffiths [5]). The method is particularly efficient for large systems of equations with diagonal dominance.

The routine D03EEF discretises a second-order equation on a two-dimensional rectangular region using finite differences and a seven-point molecule. The routine allows for cross-derivative terms, Dirichlet, Neumann or mixed boundary conditions, and either central or upwind differences. The resulting seven-diagonal difference equations are in a form suitable for passing directly to the multigrid routine D03EDF, although other solution methods could easily be used.

The routine D03FAF, based on the routine HW3CRT from FISHPACK (Swarztrauber and Sweet [8]), solves the Helmholtz equation in a three-dimensional cuboidal region, with any combination of Dirichlet, Neumann or periodic boundary conditions. The method used is based on the fast Fourier transform algorithm, and is likely to be particularly efficient on vector-processing machines.

3.2. Hyperbolic Equations

There are no routines available yet for the solution of these equations.

3.3. Parabolic Equations

There are no routines available yet for the solution of these equations.

But problems in two space dimensions plus time may be treated as a succession of elliptic equations [1], [6] using appropriate D03E- routines or one may use codes from the NAG Finite Element
3.4. Utility Routines

There are no utility routines available yet, but routines are available in the Linear Algebra Chapters for the direct and iterative solution of linear equations. Here we point to some of the routines that may be of use in solving the linear systems that arise from finite difference or finite element approximations to partial differential equation solutions. Chapters F01 and F04 should be consulted for further information and for the routine documents. Decision trees for the solution of linear systems are given in Section 3.5 of the F04 Chapter Introduction.

The following routines allow the direct solution of symmetric positive-definite systems:

- Band: F04ACF
- Variable band (skyline): F01MCF and F04MCF
- Tridiagonal: F04FAF
- Sparse: F01MAF* and F04MAF

(* the parameter DROPTL should be set to zero for F01MAF to give a direct solution)

and the following routines allow the iterative solution of symmetric positive-definite systems:

- Sparse (incomplete Cholesky): F01MAF and F04MBF

The latter routine above allows the user to supply a pre-conditioner and also allows the solution of indefinite symmetric systems.

The following routines allow the direct solution of unsymmetric systems:

- Band: F01LBF and F04LDF
- Almost block-diagonal: F01LHF and F04LHF
Tridiagonal F01LEF and F04LEF or F04EAF
Sparse F01BRF (and F01BSF) and F04AXF

and the following routine allows the iterative solution of unsymmetric systems:
Sparse F04QAF

The above routine allows the user to supply a pre-conditioner and also allows the solution of least-squares systems.

3.5. Index

Elliptic equations
   equations on rectangular grid (seven-point 2-D D03EDF
   molecule)
   discretisation on rectangular grid (seven-point 2-D D03EEF
   molecule)
   Helmholtz’s equation in three dimensions D03FAF

D03 -- Partial Differential Equations
Chapter D03

Partial Differential Equations

D03EDF Elliptic PDE, solution of finite difference equations by a multigrid technique
D03EEF Discretize a 2nd order elliptic PDE on a rectangle
D03FAF Elliptic PDE, Helmholtz equation, 3-D Cartesian coordinates

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D03EDF solves seven-diagonal systems of linear equations which
arise from the discretization of an elliptic partial differential equation on a rectangular region. This routine uses a multigrid technique.

2. Specification

```fortran
SUBROUTINE D03EDF (NGX, NGY, LDA, A, RHS, UB, MAXIT, ACC,
  1 US, U, IOUT, NUMIT, IFAIL)
  INTEGER NGX, NGY, LDA, MAXIT, IOUT, NUMIT, IFAIL
  DOUBLE PRECISION A(LDA,7), RHS(LDA), UB(NGX*NGY), ACC, US
  1 (LDA), U(LDA)
```

3. Description

D03EDF solves, by multigrid iteration, the seven-point scheme

\[
\begin{align*}
&\begin{align*}
&A_{i,j} u_{i,j} + A_{i-1,j+1} u_{i-1,j+1} + A_{i,j+1} u_{i,j+1} \\
&+ A_{i-1,j} u_{i-1,j} + A_{i,j} u_{i,j} + A_{i,j-1} u_{i,j-1} + A_{i+1,j} u_{i+1,j} \\
&+ A_1 u_{1,j} + A_{i,j} u_{i,j} = f,
\end{align*}
\end{align*}
\]

which arises from the discretization of an elliptic partial differential equation of the form

\[
(\alpha)(x,y)u_{xx} + (\beta)(x,y)u_{xy} + (\gamma)(x,y)u_{yy} + (\delta)(x,y)u_x + (\epsilon)(x,y)u_y + (\phi)(x,y)u = (\psi)(x,y)
\]

and its boundary conditions, defined on a rectangular region. This we write in matrix form as

\[
Au = f
\]

The algorithm is described in separate reports by Wesseling [2], [3] and McCarthy [1].

Systems of linear equations, matching the seven-point stencil...
defined above, are solved by a multigrid iteration. An initial estimate of the solution must be provided by the user. A zero guess may be supplied if no better approximation is available.

A 'smoother' based on incomplete Crout decomposition is used to eliminate the high frequency components of the error. A restriction operator is then used to map the system on to a sequence of coarser grids. The errors are then smoothed and prolongated (mapped onto successively finer grids). When the finest cycle is reached, the approximation to the solution is corrected. The cycle is repeated for MAXIT iterations or until the required accuracy, ACC, is reached.

D03EDF will automatically determine the number l of possible coarse grids, 'levels' of the multigrid scheme, for a particular problem. In other words, D03EDF determines the maximum integer l so that n and n can be expressed in the form

\[ n = m^2 + 1, \quad n = n^2 + 1, \text{ with } m \geq 2 \text{ and } n \geq 2. \]

It should be noted that the rate of convergence improves significantly with the number of levels used (see McCarthy [1]), so that n and n should be carefully chosen so that n - 1 and

\[ n - 1 \text{ have factors of the form } 2^x, \text{ with } l \text{ as large as possible.} \]

For good convergence the integer l should be at least 2.

D03EDF has been found to be robust in application, but being an iterative method the problem of divergence can arise. For a strictly diagonally dominant matrix A

\[ |A| > |A| \quad \text{for } k \neq 4 \]

no such problem is foreseen. The diagonal dominance of A is not a necessary condition, but should this condition be strongly violated then divergence may occur. The quickest test is to try the routine.

4. References


5. Parameters

1: NGX -- INTEGER Input
On entry: the number of interior grid points in the x-direction, n. NGX-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGX >= 3.

2: NGY -- INTEGER Input
On entry: the number of interior grid points in the y-direction, n. NGY-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGY >= 3.

3: LDA -- INTEGER Input
On entry: the first dimension of the array A as declared in the (sub)program from which D03EDF is called, which must also be a lower bound for the dimensions of the arrays RHS, US and U. It is always sufficient to set LDA>=4*(NGX*1)*(NGY*1))/3, but slightly smaller values may be permitted, depending on the values of NGX and NGY. If on entry, LDA is too small, an error message gives the minimum permitted value. (LDA must be large enough to allow space for the coarse-grid approximations).

4: A(LDA,7) -- DOUBLE PRECISION array Input/Output
On entry: A(i+(j-1)*NGX,k) must be set to A, for i = 1,2,...,NGX; j = 1,2,...,NGY and k = 1,2,...,7. On exit: A is overwritten.

5: RHS(LDA) -- DOUBLE PRECISION array Input/Output
On entry: RHS(i+(j-1)*NGX) must be set to f, for i = 1,2,...,NGX; j = 1,2,...,NGY. On exit: the first NGX*NGY elements are unchanged and the rest of the array is used as workspace.

6: UB(NGX*NGY) -- DOUBLE PRECISION array Input/Output
On entry: UB(i+(j-1)*NGX) must be set to the initial
estimate for the solution \( u_{ij} \). On exit: the corresponding component of the residual \( r = f - Au \).

7: MAXIT -- INTEGER Input
On entry: the maximum permitted number of multigrid iterations. If \( MAXIT = 0 \), no multigrid iterations are performed, but the coarse-grid approximations and incomplete Crout decompositions are computed, and may be output if IOUT is set accordingly. Constraint: \( MAXIT \geq 0 \).

8: ACC -- DOUBLE PRECISION Input
On entry: the required tolerance for convergence of the residual 2-norm:

\[
\|r\| = \sqrt{\sum_{k=1}^{NGX*NGY} \frac{r_k^2}{2}}
\]

where \( r = f - Au \) and \( u \) is the computed solution. Note that the norm is not scaled by the number of equations. The routine will stop after fewer than \( MAXIT \) iterations if the residual 2-norm is less than the specified tolerance. (If \( MAXIT > 0 \), at least one iteration is always performed.)

If on entry \( ACC = 0.0 \), then the machine precision is used as a default value for the tolerance; if \( ACC > 0.0 \), but \( ACC \) is less than the machine precision, then the routine will stop when the residual 2-norm is less than the machine precision and IFAIL will be set to 4. Constraint: \( ACC \geq 0.0 \).

9: US(LDA) -- DOUBLE PRECISION array Output
On exit: the residual 2-norm, stored in element US(1).

10: U(LDA) -- DOUBLE PRECISION array Output
On exit: the computed solution \( u \) is returned in \( U(i+(j-1)*NGX) \), for \( i = 1,2,...,NGX; j = 1,2,...,NGY \).

11: IOUT -- INTEGER Input
On entry: controls the output of printed information to the advisory message unit as returned by X04ABF:

\[ IOUT = 0 \]
No output.

\[ IOUT = 1 \]
The solution \( u_{ij} \), for \( i = 1,2,...,NGX; j = 1,2,... \)
IOUT = 2
The residual 2-norm after each iteration, with the
reduction factor over the previous iteration.

IOUT = 3
As for IOUT = 1 and IOUT = 2.

IOUT = 4
As for IOUT = 3, plus the final residual (as returned
in UB).

IOUT = 5
As for IOUT = 4, plus the initial elements of A and
RHS.

IOUT = 6
As for IOUT = 5, plus the Galerkin coarse grid
approximations.

IOUT = 7
As for IOUT = 6, plus the incomplete Crout
decompositions.

IOUT = 8
As for IOUT = 7, plus the residual after each
iteration.
The elements $A(p,k)$, the Galerkin coarse grid approximations
and the incomplete Crout decompositions are output in the
format:

$$
\text{Y-index} = j \\
\text{X-index} = i \ A(p,1) \ A(p,2) \ A(p,3) \ A(p,4) \ A(p,5) \ A(p,6) \\
\hspace{1cm} A(p,7)
$$

where $p = i + (j-1) \times NGX$, $i = 1,2,...,NGX$ and $j = 1,2,...
,NGY$.
The vectors $U(p)$, $UB(p)$, $RHS(p)$ are output in matrix form
with NGY rows and NGX columns. Where NGX > 10, the NGX
values for a given j-value are produced in rows of 10.
Values of IOUT > 4 may yield considerable amounts of output.
Constraint: $0 \leq \text{IOUT} \leq 8$.

12: NUMIT -- INTEGER Output
On exit: the number of iterations performed.

13: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction: the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1

On entry NGX < 3,

or NGY < 3,

or LDA is too small,

or ACC < 0.0,

or MAXIT < 0,

or IOUT < 0,

or IOUT > 8.

IFAIL = 2

MAXIT iterations have been performed with the residual 2-norm decreasing at each iteration but the residual 2-norm has not been reduced to less than the specified tolerance (see ACC). Examine the progress of the iteration by setting IOUT >= 2.

IFAIL = 3

As for IFAIL = 2, except that at one or more iterations the residual 2-norm did not decrease. It is likely that the method fails to converge for the given matrix A.

IFAIL = 4

On entry ACC is less than the machine precision. The routine terminated because the residual norm is less than the machine precision.

7. Accuracy

See ACC (Section 5).

8. Further Comments
The rate of convergence of this routine is strongly dependent upon the number of levels, \( l \), in the multigrid scheme, and thus the choice of \( \text{NGX} \) and \( \text{NGY} \) is very important. The user is advised to experiment with different values of \( \text{NGX} \) and \( \text{NGY} \) to see the effect they have on the rate of convergence; for example, using a value such as \( \text{NGX} = 65 \) (\( =2 +1 \)) followed by \( \text{NGX} = 64 \) (for which \( l = 1 \)).

9. Example

The program solves the elliptic partial differential equation

\[
-\alpha U_{xx} + U_{xy} + U_{yy} = -4, \quad \alpha = 1.7
\]

on the unit square \( 0 \leq x, y \leq 1 \), with boundary conditions

\[
\begin{align*}
\{x=0, & \quad (0 \leq y \leq 1) \\
U=0 & \quad \{y=0, \quad (0 \leq x \leq 1) \\
U=1 & \quad \{x=1, \quad (0 \leq y \leq 1) \\
\{y=1, & \quad (0 \leq x \leq 1)
\end{align*}
\]

For the equation to be elliptic, \( \alpha \) must be less than 2.

The equation is discretized on a square grid with mesh spacing \( h \) in both directions using the following approximations:

Please see figure in printed Reference Manual

\[
\begin{align*}
1 \quad U &= -(U_{xx} - 2U + U) \\
& \quad \text{xx 2 E 0 W} \\
& \quad h
\end{align*}
\]

\[
\begin{align*}
1 \quad U &= -(U_{yy} - 2U + U) \\
& \quad \text{yy 2 N 0 S} \\
& \quad h
\end{align*}
\]

\[
\begin{align*}
1 \quad U &= -(U_{xy} - 2U + U - 2U + U) \\
& \quad \text{xy 2 N NW E 0 W SE S} \\
& \quad 2h
\end{align*}
\]

Thus the following equations are solved:
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose
D03EEF discretizes a second order elliptic partial differential equation (PDE) on a rectangular region.

2. Specification

```fortran
SUBROUTINE D03EEF (XMIN, XMAX, YMIN, YMAX, PDEF, BNDY,
  NGX, NGY, LDA, A, RHS, SCHEME, IFAIL)
  INTEGER NGX, NGY, LDA, IFAIL
  DOUBLE PRECISION XMIN, XMAX, YMIN, YMAX, A(LDA,7), RHS(LDA)
  CHARACTER*1 SCHEME
  EXTERNAL PDEF, BNDY
```

3. Description
D03EEF discretizes a second order linear elliptic partial differential equation of the form

\[-(\alpha)u + (1- \alpha)u + (1- \alpha)u + (-4+\alpha)u + (1- \alpha)u + \alpha u + \alpha u = -4h\]
\begin{align*}
\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \left( \frac{\partial}{\partial x} \right)^2 \left( \frac{\partial}{\partial y} \right)^2 U &= \left( \frac{\partial}{\partial x} \right)^2 \left( \frac{\partial}{\partial y} \right)^2 \left[ \left( \frac{\partial}{\partial x} \right)^2 + \left( \frac{\partial}{\partial y} \right)^2 \right] U \\
&= \left( \frac{\partial}{\partial x} \right)^2 \frac{\partial^2 U}{\partial x^2} + \left( \frac{\partial}{\partial y} \right)^2 \frac{\partial^2 U}{\partial y^2} \quad \text{(1)}
\end{align*}

on a rectangular region

\begin{align*}
x \leq x \leq x \\
A & \quad B
\end{align*}

\begin{align*}
y \leq y \leq y \\
A & \quad B
\end{align*}

subject to boundary conditions of the form

\begin{align*}
\frac{\partial U}{\partial n} = a(x,y)U + b(x,y) &= c(x,y) \\
\frac{\partial U}{\partial n}
\end{align*}

where \( \frac{\partial}{\partial n} \) denotes the outward pointing normal derivative on the boundary. Equation (1) is said to be elliptic if

\[ 2 \quad 4(\alpha)(x,y)(\gamma)(x,y) \geq (\beta)(x,y) \]

for all points in the rectangular region. The linear equations produced are in a form suitable for passing directly to the multigrid routine D03EDF.

The equation is discretized on a rectangular grid, with \( n \) grid points in the \( x \)-direction and \( n \) grid points in the \( y \)-direction.

The grid spacing used is therefore

\begin{align*}
h &= (x_B - x_A)/(n - 1) \\
h &= (y_B - y_A)/(n - 1)
\end{align*}

and the co-ordinates of the grid points \((x_i, y_j)\) are

\begin{align*}
\text{and the co-ordinates of the grid points } (x_i, y_j) \text{ are}
\end{align*}
\[ x = x + (i-1)h, \quad i = 1, 2, \ldots, n, \]
\[ y = y + (j-1)h, \quad j = 1, 2, \ldots, n. \]

At each grid point \((x_i, y_j)\) six neighbouring grid points are used to approximate the partial differential equation, so that the equation is discretized on the following seven-point stencil:

Please see figure in printed Reference Manual.

For convenience the approximation \(u_{ij}\) to the exact solution \(U(x_i, y_j)\) is denoted by \(u_{ij}\), and the neighbouring approximations are labelled according to points of the compass as shown. Where numerical labels for the seven points are required, these are also shown above.

The following approximations are used for the second derivatives:

\[ \frac{\partial^2 U}{\partial x^2} = \frac{1}{2h^2} (u_{i+1,j} - 2u_{i,j} + u_{i-1,j}). \]
\[ \frac{\partial^2 U}{\partial y^2} = \frac{1}{2h^2} (u_{i,j+1} - 2u_{i,j} + u_{i,j-1}). \]
\[ \frac{\partial^2 U}{\partial x \partial y} = \frac{1}{2h^2} (u_{i+1,j+1} - u_{i+1,j} - u_{i,j+1} + u_{i,j}). \]

Two possible schemes may be used to approximate the first derivatives:

Central Differences.
Upwind Differences

\[
\frac{dU}{dx} = \begin{cases} 
\frac{U_{i+1}^n - U_i^n}{h} & \text{if } \delta(x,y) > 0 \\
\frac{U_i^n - U_{i-1}^n}{h} & \text{if } \delta(x,y) < 0 
\end{cases}
\]

\[
\frac{dU}{dy} = \begin{cases} 
\frac{U_{j+1}^n - U_j^n}{h} & \text{if } \epsilon(x,y) > 0 \\
\frac{U_j^n - U_{j-1}^n}{h} & \text{if } \epsilon(x,y) < 0 
\end{cases}
\]

Central differences are more accurate than upwind differences, but upwind differences may lead to a more diagonally dominant matrix for those problems where the coefficients of the first derivatives are significantly larger than the coefficients of the second derivatives.

The approximations used for the first derivatives may be written in a more compact form as follows:

\[
\frac{dU}{dx} = \frac{1}{2h} \left( (k-1)u - 2k u + (k+1)u \right)
\]

\[
\frac{dU}{dy} = \frac{1}{2h} \left( (k-1)u - 2k u + (k+1)u \right)
\]
where \( k = \text{sign} (\delta) \) and \( k = \text{sign} (\epsilon) \) for upwind differences, and \( k = k = 0 \) for central differences.

At all points in the rectangular domain, including the boundary, the coefficients in the partial differential equation are evaluated by calling the user-supplied subroutine PDEF, and applying the approximations. This leads to a seven-diagonal system of linear equations of the form:

\[
\begin{align*}
6 & \quad 7 \\
A_{ij} u_{i-1,j+1} & + A_{ij} u_{ij,j+1} \\
3 & \quad 4 & 5 \\
+ A_{ij} u_{i-1,j} & + A_{ij} u_{ij,j} + A_{ij} u_{ij,i+1,j} \\
1 & \quad 2 \\
+ A_{ij} u_{i,j-1} & + A_{ij} u_{ij,i+1,j-1} = f, \quad i=1,2,\ldots,n; \quad j=1,2,\ldots,n,
\end{align*}
\]

where the coefficients are given by

\[
\begin{align*}
A_{ij} & = (\beta)(x_i, y_j) - \frac{1}{h_x} + (\gamma)(x_i, y_j) - \frac{1}{h_y} + (\epsilon)(x_i, y_j) - \frac{(k-1)}{h_y} \\
A_{ij} & = -(\beta)(x_i, y_j) - \frac{1}{h_x} \\
A_{ij} & = (\alpha)(x_i, y_j) + (\beta)(x_i, y_j) - \frac{1}{h_x} + (\delta)(x_i, y_j) - \frac{(k-1)}{h_x} \\
A_{ij} & = -(\alpha)(x_i, y_j) - \frac{1}{h_x} - (\beta)(x_i, y_j) - \frac{1}{h_y} - (\gamma)(x_i, y_j) - \frac{(k-1)}{h_y} \\
A_{ij} & = -(\delta)(x_i, y_j) - \frac{1}{h_x} - (\epsilon)(x_i, y_j) - \frac{(k-1)}{h_y} - (\phi)(x_i, y_j) - \frac{(k-1)}{h_y} \\
A_{ij} & = - (\gamma)(x_i, y_j) - \frac{(k-1)}{h_y}
\end{align*}
\]
These equations then have to be modified to take account of the boundary conditions. These may be Dirichlet (where the solution is given), Neumann (where the derivative of the solution is given), or mixed (where a linear combination of solution and derivative is given).

If the boundary conditions are Dirichlet, there are an infinity of possible equations which may be applied:

\[(\mu)u = (\mu)f \quad , \quad (\mu)/=0.\]

If D03EDF is used to solve the discretized equations, it turns out that the choice of \((\mu)\) can have a dramatic effect on the rate of convergence, and the obvious choice \((\mu)=1\) is not the best. Some choices may even cause the multigrid method to fail altogether. In practice it has been found that a value of the same order as the other diagonal elements of the matrix is best, and the following value has been found to work well in practice:

\[(\mu)=\min(\{\begin{array}{c} 2 \\ 2 \\ 4 \end{array}\}, A ).\]

If the boundary conditions are either mixed or Neumann (i.e., B


\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \text{ on return from the user-supplied subroutine BNDY), then one of the points in the seven-point stencil lies outside the domain. In this case the normal derivative in the boundary conditions is used to eliminate the 'fictitious' point, } u : \]

\[ \begin{align*}
\frac{\partial u}{\partial n} &= \frac{u_{\text{outside}} - u_{\text{inside}}}{2h}, \\
\end{align*} \tag{3} \]

It should be noted that if the boundary conditions are Neumann and \((\phi)(x,y)=0\), then there is no unique solution. The routine returns with IFAIL = 5 in this case, and the seven-diagonal matrix is singular.

The four corners are treated separately. The user-supplied subroutine BNDY is called twice, once along each of the edges meeting at the corner. If both boundary conditions at this point are Dirichlet and the prescribed solution values agree, then this value is used in an equation of the form (2). If the prescribed solution is discontinuous at the corner, then the average of the two values is used. If one boundary condition is Dirichlet and the other is mixed, then the value prescribed by the Dirichlet condition is used in an equation of the form given above. Finally, if both conditions are mixed or Neumann, then two 'fictitious' points are eliminated using two equations of the form (3).

It is possible that equations for which the solution is known at all points on the boundary, have coefficients which are not defined on the boundary. Since this routine calls the user-supplied subroutine PDEF at all points in the domain, including boundary points, arithmetic errors may occur in the user's routine PDEF which this routine cannot trap. If the user has an equation with Dirichlet boundary conditions (i.e., \(B = 0\) at all points on the boundary), but with PDE coefficients which are singular on the boundary, then D03EDF could be called directly only using interior grid points with the user's own discretization.

After the equations have been set up as described above, they are checked for diagonal dominance. That is to say,

\[ \frac{4}{|A_{ij}|} > \sum_{k=1}^{4} |A_{ij}|, \quad i=1,2,...,n; \quad j=1,2,...,n. \]

If this condition is not satisfied then the routine returns with IFAIL = 6. The multigrid routine D03EDF may still converge in
this case, but if the coefficients of the first derivatives in
the partial differential equation are large compared with the
coefficients of the second derivative, the user should consider
using upwind differences (SCHEME = 'U').

Since this routine is designed primarily for use with D03EDF,
this document should be read in conjunction with the document for
that routine.

4. References

Method. Multigrid Methods. Lecture Notes in Mathematics. 960
Springer-Verlag. 614--630.

5. Parameters

1: XMIN -- DOUBLE PRECISION Input
2: XMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper x co-ordinates of the
   rectangular region respectively, x and x . Constraint: XMIN
   A B
   < XMAX.
3: YMIN -- DOUBLE PRECISION Input
4: YMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper y co-ordinates of the
   rectangular region respectively, y and y . Constraint: YMIN
   A B
   < YMAX.
5: PDEF -- SUBROUTINE, supplied by the user.
   External Procedure
   PDEF must evaluate the functions (alpha)(x,y), (beta)(x,y),
   (gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y) and
   (psi)(x,y) which define the equation at a general point
   (x,y).
   Its specification is:

   SUBROUTINE PDEF (X, Y, ALPHA, BETA, GAMMA,
   1 DELTA, EPSILON, PHI, PSI)
   DOUBLE PRECISION X, Y, ALPHA, BETA, GAMMA, DELTA,
   1 EPSILON, PHI, PSI

   1: X -- DOUBLE PRECISION Input
   2: Y -- DOUBLE PRECISION Input
On entry: the x and y co-ordinates of the point at which the coefficients of the partial differential equation are to be evaluated.

3: ALPHA -- DOUBLE PRECISION Output
4: BETA -- DOUBLE PRECISION Output
5: GAMMA -- DOUBLE PRECISION Output
6: DELTA -- DOUBLE PRECISION Output
7: EPSLON -- DOUBLE PRECISION Output
8: PHI -- DOUBLE PRECISION Output
9: PSI -- DOUBLE PRECISION Output

On exit: ALPHA, BETA, GAMMA, DELTA, EPSLON, PHI and PSI must be set to the values of (alpha)(x,y), (beta)(x,y), (gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y) and (psi)(x,y) respectively at the point specified by X and Y.

PDEF must be declared as EXTERNAL in the (sub)program from which D03EEF is called. Parameters denoted as Input must not be changed by this procedure.

6: BNDY -- SUBROUTINE, supplied by the user.

External Procedure
BNDY must evaluate the functions a(x,y), b(x,y), and c(x,y) involved in the boundary conditions.

Its specification is:

```
SUBROUTINE BNDY (X, Y, A, B, C, IBND)
INTEGER IBND
DOUBLE PRECISION X, Y, A, B, C
```

1: X -- DOUBLE PRECISION Input
2: Y -- DOUBLE PRECISION Input

On entry: the x and y co-ordinates of the point at which the boundary conditions are to be evaluated.

3: A -- DOUBLE PRECISION Output
4: B -- DOUBLE PRECISION Output
5: C -- DOUBLE PRECISION Output

On exit: A, B and C must be set to the values of the functions appearing in the boundary conditions.
6: IBND -- INTEGER
   On entry: specifies on which boundary the point (X,Y) lies. IBND = 0, 1, 2 or 3 according as the point lies on the bottom, right, top or left boundary.
   BNDY must be declared as EXTERNAL in the (sub)program from which D03EEF is called. Parameters denoted as Input must not be changed by this procedure.

7: NGX -- INTEGER

8:NGY -- INTEGER
   On entry: the number of interior grid points in the x- and y-directions respectively, n and n. If the seven-diagonal equations are to be solved by D03EDF, then NGX-1 and NGY-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGX >= 3, NGY >= 3.

9: LDA -- INTEGER
   On entry: the first dimension of the array A as declared in the (sub)program from which D03EEF is called.
   Constraint: if only the seven-diagonal equations are required, then LDA >= NGX*NGY. If a call to this routine is to be followed by a call to D03EDF to solve the seven-diagonal linear equations, LDA >= (4*(NGX+1)*(NGY+1))/3.
   Note: this routine only checks the former condition. D03EDF, if called, will check the latter condition.

10: A(LDA,7) -- DOUBLE PRECISION array
   On exit: A(i,j), for i=1,2,...,NGX*NGY; j = 1,2,...,7, contains the seven-diagonal linear equations produced by the discretization described above. If LDA > NGX*NGY, the remaining elements are not referenced by the routine, but if LDA >= (4*(NGX+1)*(NGY+1))/3 then the array A can be passed directly to D03EDF, where these elements are used as workspace.

11: RHS(LDA) -- DOUBLE PRECISION array
   On exit: the first NGX*NGY elements contain the right-hand sides of the seven-diagonal linear equations produced by the discretization described above. If LDA > NGX*NGY, the remaining elements are not referenced by the routine, but if LDA >= (4*(NGY+1)*(NGY+1))/3 then the array RHS can be passed directly to D03EDF, where these elements are used as workspace.

12: SCHEME -- CHARACTER*1
   Input
On entry: the type of approximation to be used for the first derivatives which occur in the partial differential equation.

If SCHEME = 'C', then central differences are used.

If SCHEME = 'U', then upwind differences are used.
Constraint: SCHEME = 'C' or 'U'.

Note: generally speaking, if at least one of the coefficients multiplying the first derivatives (DELTA or EPSLON as returned by PDEF) are large compared with the coefficients multiplying the second derivatives, then upwind differences may be more appropriate. Upwind differences are less accurate than central differences, but may result in more rapid convergence for strongly convective equations. The easiest test is to try both schemes.

13: IFAIL -- INTEGER          Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry XMIN >= XMAX,
or YMIN >= YMAX,
or NGX < 3,
or NGY < 3,
or LDA < NGX*NGY,
or SCHEME is not one of 'C' or 'U'.
CHAPTER 15. CHAPTER N

IFAIL= 2
At some point on the boundary there is a derivative in the boundary conditions (B /= 0 on return from a BNDY) and there is a non-zero coefficient of the mixed derivative (BETA /= 0 on return from PDEF).

IFAIL= 3
A null boundary has been specified, i.e., at some point both A and B are zero on return from a call to BNDY.

IFAIL= 4
The equation is not elliptic, i.e., 4*ALPHA*GAMMA<BETA after a call to PDEF. The discretization has been completed, but the convergence of D03EDF cannot be guaranteed.

IFAIL= 5
The boundary conditions are purely Neumann (only the derivative is specified) and there is, in general, no unique solution.

IFAIL= 6
The equations were not diagonally dominant. (See Section 3).

7. Accuracy
Not applicable.

8. Further Comments
If this routine is used as a pre-processor to the multigrid routine D03EDF it should be noted that the rate of convergence of that routine is strongly dependent upon the number of levels in the multigrid scheme, and thus the choice of NGX and NGY is very important.

9. Example
The program solves the elliptic partial differential equation

\[
\begin{align*}
\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + 50 \left( \frac{\partial U}{\partial x} \right) &= f(x,y) \\
\frac{\partial U}{\partial x} &= g(x,y)
\end{align*}
\]
on the unit square 0≤x, y≤1, with boundary conditions

\[ \frac{dU}{dx} \text{ given on } x=0 \text{ and } y=0, \]
\[ \frac{dU}{dn} \text{ given on } x=1 \text{ and } y=1. \]

The function \( f(x, y) \) and the exact form of the boundary conditions are derived from the exact solution \( U(x, y) = \sin x \sin y \).

The equation is first solved using central differences. Since the coefficients of the first derivatives are large, the linear equations are not diagonally dominated, and convergence is slow. The equation is solved a second time with upwind differences, showing that convergence is more rapid, but the solution is less accurate.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D03 -- Partial Differential Equations
D03FAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D03FAF solves the Helmholtz equation in Cartesian co-ordinates in three dimensions using the standard seven-point finite difference approximation. This routine is designed to be particularly efficient on vector processors.

2. Specification

SUBROUTINE D03FAF (XS, XF, L, LBDCND, BDXS, BDXF, YS, YF, 1 M, MBDCND, BDYS, BDYF, ZS, ZF, N, 2 NBDCND, BDZS, BDZF, LAMBDA, LDIMF, 3 MDIMF, F, PERTRB, W, LWRK, IFAIL)

INTEGER L, LBDCND, M, MBDCND, N, NBDCND, LDIMF, 1 MDIMF, LWRK, IFAIL
DOUBLE PRECISION XS, XF, BDXS(MDIMF,N+1), BDXF(MDIMF,N+1), 1 YS, YF, BDYS(LDIMF,N+1), BDYF(LDIMF,N+1), 2 ZS, ZF, BDZS(LDIMF,M+1), BDZF(LDIMF,M+1),
3. Description

D03FAF solves the three-dimensional Helmholtz equation in cartesian co-ordinates:

\[
\begin{align*}
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \lambda u &= f(x,y,z) \\
\end{align*}
\]

This subroutine forms the system of linear equations resulting from the standard seven-point finite difference equations, and then solves the system using a method based on the fast Fourier transform (FFT) described by Swarztrauber [1]. This subroutine is based on the routine HW3CRT from FISHPACK (see Swarztrauber and Sweet [2]).

More precisely, the routine replaces all the second derivatives by second-order central difference approximations, resulting in a block tridiagonal system of linear equations. The equations are modified to allow for the prescribed boundary conditions. Either the solution or the derivative of the solution may be specified on any of the boundaries, or the solution may be specified to be periodic in any of the three dimensions. By taking the discrete Fourier transform in the x- and y-directions, the equations are reduced to sets of tridiagonal systems of equations. The Fourier transforms required are computed using the multiple FFT routines found in Chapter C06 of the NAG Fortran Library.

4. References


5. Parameters

1: XS -- DOUBLE PRECISION Input
On entry: the lower bound of the range of x, i.e., XS <= x <= XF. Constraint: XS < XF.

2: XF -- DOUBLE PRECISION Input
On entry: the upper bound of the range of \( x \), i.e., \( XS <= x <= XF \). Constraint: \( XS < XF \).

3: \( L \) -- INTEGER Input
On entry: the number of panels into which the interval \((XS,XF)\) is subdivided. Hence, there will be \( L+1 \) grid points in the \( x \)-direction given by \( x = XS + (i-1) \times (delta)x \), for \( i = 1, 2, \ldots, L+1 \), where \( (delta)x = (XF - XS) / L \) is the panel width. Constraint: \( L >= 5 \).

4: \( LBDCND \) -- INTEGER Input
On entry: indicates the type of boundary conditions at \( x = XS \) and \( x = XF \).
\( LBDCND = 0 \) if the solution is periodic in \( x \), i.e.,
\[ u(XS,y,z) = u(XF,y,z) \]
\( LBDCND = 1 \) if the solution is specified at \( x = XS \) and \( x = XF \).
\( LBDCND = 2 \) if the solution is specified at \( x = XS \) and the derivative of the solution with respect to \( x \) is specified at \( x = XF \).
\( LBDCND = 3 \) if the derivative of the solution with respect to \( x \) is specified at \( x = XS \) and \( x = XF \).
\( LBDCND = 4 \) if the derivative of the solution with respect to \( x \) is specified at \( x = XS \) and the solution is specified at \( x = XF \).
Constraint: \( 0 <= LBDCND <= 4 \).

5: \( BDXS(MDIMF,N+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( x \) at \( x = XS \). When \( LBDCND = 3 \) or \( 4 \), \( BDXS(j,k) = u(XS,y,z) \), for \( j=1,2,\ldots,M+1; k=1,2,\ldots,N+1 \).
When \( LBDCND \) has any other value, \( BDXS \) is not referenced.

6: \( BDXF(MDIMF,N+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( x \) at \( x = XF \). When \( LBDCND = 2 \) or \( 3 \), \( BDXF(j,k) = u(XF,y,z) \), for \( j=1,2,\ldots,M+1; k=1,2,\ldots,N+1 \).
When LBDCND has any other value, BDXF is not referenced.

7:  YS -- DOUBLE PRECISION  
Input  
On entry: the lower bound of the range of y, i.e., YS <= y <= YF. Constraint: YS < YF.

8:  YF -- DOUBLE PRECISION  
Input  
On entry: the upper bound of the range of y, i.e., YS <= y <= YF. Constraint: YS < YF.

9:  M -- INTEGER  
Input  
On entry: the number of panels into which the interval (YS,YF) is subdivided. Hence, there will be M+1 grid points in the y-direction given by 
\[ y = YS + (j-1)*(\Delta y) \]  
j = 1,2,...,M+1, where (\Delta y) = (YF-YS)/M is the panel width. Constraint: M >= 5.

10: MBDCND -- INTEGER  
Input  
On entry: indicates the type of boundary conditions at y = YS and y = YF.  
MBDCND = 0  
if the solution is periodic in y, i.e., u(x,YF,z)=u(x,YS,z).

MBDCND = 1  
if the solution is specified at y = YS and y = YF.

MBDCND = 2  
if the solution is specified at y = YS and the derivative of the solution with respect to y is specified at y = YF.

MBDCND = 3  
if the derivative of the solution with respect to y is specified at y = YS and y = YF.

MBDCND = 4  
if the derivative of the solution with respect to y is specified at y = YS and the solution is specified at y = YF.  
Constraint: 0 <= MBDCND <= 4.

11: BDYS(LDIMF,N+1) -- DOUBLE PRECISION array  
Input  
On entry: the values of the derivative of the solution with respect to y at y = YS. When MBDCND = 3 or 4, BDYS (i,k)=u (x ,y ,z ), for i=1,2,...,L+1; k=1,2,...,N+1.  
y i s k

When MBDCND has any other value, BDYS is not referenced.
12: BDYF(LDIMF,N+1) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to y at y = YF. When MBDCND = 2 or 3, BDYF (i,k)=u (x ,YF,z ), for i=1,2,...,L+1; k=1,2,...,N+1.
y i k
When MBDCND has any other value, BDYF is not referenced.

13: ZS -- DOUBLE PRECISION Input
On entry: the lower bound of the range of z, i.e., ZS <= z <= ZF. Constraint: ZS < ZF.

14: ZF -- DOUBLE PRECISION Input
On entry: the upper bound of the range of z, i.e., ZS <= z <= ZF. Constraint: ZS < ZF.

15: N -- INTEGER Input
On entry: the number of panels into which the interval (ZS,ZF) is subdivided. Hence, there will be N+1 grid points in the z-direction given by z =ZS+(k-1)*(delta)z, for
k=1,2,...,N+1, where (delta)z=(ZF-ZS)/N is the panel width. Constraint: N >= 5.

16: NBDCND -- INTEGER Input
On entry: specifies the type of boundary conditions at z = ZS and z = ZF.
NBDCND = 0
if the solution is periodic in z, i.e.,
u(x,y,ZF)=u(x,y,ZS).

NBDCND = 1
if the solution is specified at z = ZS and z = ZF.

NBDCND = 2
if the solution is specified at z = ZS and the derivative of the solution with respect to z is specified at z = ZF.

NBDCND = 3
if the derivative of the solution with respect to z is specified at z = ZS and z = ZF.

NBDCND = 4
if the derivative of the solution with respect to z is specified at z = ZS and the solution is specified at z = ZF.
Constraint: 0 <= NBDCND <= 4.
17: BDZS(LDIMF,M+1) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to z at z = ZS. When NBDCND = 3 or 4, BDZS
\( (i,j)=u(x,y,ZS)=u(x,y,z) \), for \( i=1,2,...,L+1; \)
\( j=1,2,...,M+1. \)
When NBDCND has any other value, BDZS is not referenced.

18: BDZF(LDIMF,M+1) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to z at z = ZF. When NBDCND = 2 or 3, BDZF
\( (i,j)=u(x,y,ZF)=u(x,y,z) \), for \( i=1,2,...,L+1; \)
\( j=1,2,...,M+1. \)
When NBDCND has any other value, BDZF is not referenced.

19: LAMBDA -- DOUBLE PRECISION Input
On entry: the constant (\( \lambda \)) in the Helmholtz equation. For certain positive values of (\( \lambda \)) a solution to the differential equation may not exist, and close to these values the solution of the discretized problem will be extremely ill-conditioned. If (\( \lambda > 0 \)), then D03FAF will set IFAIL to 3, but will still attempt to find a solution. However, since in general the values of (\( \lambda \)) for which no solution exists cannot be predicted a priori, the user is advised to treat any results computed with (\( \lambda > 0 \)) with great caution.

20: LDIMF -- INTEGER Input
On entry: the first dimension of the arrays F, BDYS, BDYF, BDZS and BDZF as declared in the (sub)program from which D03FAF is called.
Constraint: LDIMF >= L + 1.

21: MDIMF -- INTEGER Input
On entry: the second dimension of the array F and the first dimension of the arrays BDXS and BDXF as declared in the (sub)program from which D03FAF is called.
Constraint: MDIMF >= M + 1.

22: F(LDIMF,MDIMF,N+1) -- DOUBLE PRECISION array Input/Output
On entry: the values of the right-side of the Helmholtz equation and boundary values (if any).
\( F(i,j,k)=f(x,y,z) \) for \( i=2,3,...,L; j=2,3,...,M \) and \( k=2,3,...,N. \)
On the boundaries $F$ is defined by

**LBDCND**

$F(1,j,k) \quad F(L+1,j,k)$

1. $f(XS,y,z) f(XS,y,z)$
   $j \quad k \quad j \quad k$

2. $u(XS,y,z) u(XF,y,z)$
   $j \quad k \quad j \quad k$

3. $u(XS,y,z) f(XF,y,z)$
   $j=1,2,...,M+1$
   $j \quad k \quad j \quad k$

4. $f(XS,y,z) u(XF,y,z)$
   $j \quad k \quad j \quad k$

**MBDCND**

$F(i,1,k) \quad F(i,M+1,k)$

1. $f(x,YS,z) f(x,YS,z)$
   $i \quad k \quad i \quad k$

2. $u(x,YS,z) u(x,YF,z)$
   $i \quad k \quad i \quad k$

3. $u(x,YS,z) f(x,YF,z)$
   $i=1,2,...,L+1$
   $i \quad k \quad i \quad k$

4. $f(x,YS,z) u(x,YF,z)$
   $i \quad i \quad k \quad i \quad k$

**NBDCND**

$F(i,j,1) \quad F(i,j,N+1)$

1. $f(x,y, ZS) f(x,y, ZS)$
   $i \quad j \quad i \quad j$

2. $u(x,y, ZS) u(x,y, ZF)$
   $i \quad j \quad i \quad j$
2 \[ u(x_i, y_j, ZS_i) f(x_i, y_j, ZF_i) \quad i=1,2,\ldots,L+1 \]
\[ j=1,2,\ldots,M+1 \]

3 \[ f(x_i, y_j, ZS_i) f(x_i, y_j, ZF_i) \]
\[ j=1,2,\ldots,M+1 \]

4 \[ f(x_i, y_j, ZS_i) u(x_i, y_j, ZF_i) \]
\[ j=1,2,\ldots,M+1 \]

Note: if the table calls for both the solution \( u \) and the right-hand side \( f \) on a boundary, then the solution must be specified. On exit: \( F \) contains the solution \( u(i,j,k) \) of the finite difference approximation for the grid point \( (x_i, y_j, z_k) \) for \( i=1,2,\ldots,L+1 \), \( j=1,2,\ldots,M+1 \) and \( k=1,2,\ldots,N+1 \).

23: \textbf{PERTRB} -- DOUBLE PRECISION \hspace{1cm} Output
On exit: \( \text{PERTRB} = 0 \), unless a solution to Poisson’s equation \( ((\lambda)=0) \) is required with a combination of periodic or derivative boundary conditions \( (\text{LBCND}, \text{MBCND} \text{ and } \text{NBDCND} = 0 \text{ or } 3) \). In this case a solution may not exist. \( \text{PERTRB} \) is a constant, calculated and subtracted from the array \( F \), which ensures that a solution exists. \text{D03FAF} then computes this solution, which is a least-squares solution to the original approximation. This solution is not unique and is unnormalised. The value of \( \text{PERTRB} \) should be small compared to the right-hand side \( F \), otherwise a solution has been obtained to an essentially different problem. This comparison should always be made to insure that a meaningful solution has been obtained.

24: \textbf{W(LWRK)} -- DOUBLE PRECISION array \hspace{1cm} Workspace
25: \textbf{LWRK} -- INTEGER \hspace{1cm} Input
On entry: the dimension of the array \( W \) as declared in the (sub)program from which \text{D03FAF} is called. \( \text{LWRK} > 2 \ast (N+1) \ast \max(L,M) + 3 \ast N + 4 \ast N + 6 \) is an upper bound on the required size of \( W \). If \( \text{LWRK} \) is too small, the routine exits with \( \text{IFAIL} = 2 \), and if on entry \( \text{IFAIL} = 0 \) or \( \text{IFAIL} = -1 \), a message is output giving the exact value of \( \text{LWRK} \) required to solve the current problem.

26: \textbf{IFAIL} -- INTEGER \hspace{1cm} Input/Output
On entry: \( \text{IFAIL} \) must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: \( \text{IFAIL} = 0 \) unless the routine detects an error or gives a warning (see Section 6).
For this routine, because the values of output parameters may be useful even if IFAIL ≠ 0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry XS >= XF,
  or L < 5,
  or LBDCND < 0,
  or LBDCND > 4,
  or YS >= YF,
  or M < 5,
  or MBDCND < 0,
  or MBDCND > 4,
  or ZS >= ZF,
  or N < 5,
  or NBDCND < 0,
  or NBDCND > 4,
  or LDIMF < L + 1 > 0,
  or MDIMF < M + 1.

IFAIL= 2
On entry LWRK is too small.

IFAIL= 3
On entry (lambda) > 0.

7. Accuracy

None.
8. Further Comments

The execution time is roughly proportional to $L^2M^2N^2(\log L+\log M+5)$, but also depends on input parameters $LBDCND$ and $MBDCND$.

9. Example

The example solves the Helmholtz equation

$$
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + (\lambda)u = f(x,y,z)
$$

for $(x,y,z)$ is in $[0,1] \times [0,2\pi] \times [0,\pi]$ where $(\lambda) = -2$, and $f(x,y,z)$ is derived from the exact solution

$$
u(x,y,z) = x \sin(y) \cos(z).
$$

The equation is subject to the following boundary conditions, again derived from the exact solution given above.

- $u(0,y,z)$ and $u(1,y,z)$ are prescribed (i.e., $LBDCND = 1$).
- $u(x,0,z) = u(x,2\pi,z)$ (i.e., $MBDCND = 0$).
- $u(x,y,0)$ and $u(x,y,\pi)$ are prescribed (i.e., $NBDCND = 2$).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagPartialDifferentialEquationsPackage (NAGD03)

Exports:
  d03edf  d03eef  d03faf

--- package NAGD03 NagPartialDifferentialEquationsPackage ---

)abbrev package NAGD03 NagPartialDifferentialEquationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:51 1994
++ Description:
++ This package uses the NAG Library to solve partial
++ differential equations.

NagPartialDifferentialEquationsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
  d03edf : (Integer,Integer,Integer,Integer,DoubleFloat,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
  ++ d03edf(ngx,ngy,lda,maxit,acc,iout,a,rhs,ub,ifail)
  ++ solves seven-diagonal systems of linear equations which
  ++ arise from the discretization of an elliptic partial differential
  ++ equation on a rectangular region. This routine uses a multigrid
  ++ technique.
  ++ See \downlink{Manual Page}{manpageXXd03edf}.

  d03eef : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,String,Integer,Union(fn:FileName,fp:Asp73(PDEF)),Union(fn:FileName,fp:Asp74(BNDY))) -> Result
  ++ d03eef(xmin,xmax,ymin,ymax,ngx,ngy,lda,scheme,ifail,pdef,bndy)
  ++ discretizes a second order elliptic partial differential
  ++ equation (PDE) on a rectangular region.
  ++ See \downlink{Manual Page}{manpageXXd03eef}.
CHAPTER 15. CHAPTER N

\[ d03faf : (\text{DoubleFloat}, \text{DoubleFloat}, \text{Integer}, \text{Integer}, \_
\text{Matrix DoubleFloat}, \text{Matrix DoubleFloat}, \text{DoubleFloat}, \text{DoubleFloat}, \_
\text{Integer}, \text{Integer}, \text{Matrix DoubleFloat}, \text{Matrix DoubleFloat}, \_
\text{DoubleFloat}, \text{Integer}, \text{Integer}, \text{Matrix DoubleFloat}, \text{Matrix DoubleFloat}, \_
\text{DoubleFloat}, \text{Integer}, \text{Integer}, \text{Integer}, \_
\text{ThreeDimensionalMatrix DoubleFloat}, \text{Integer}) \rightarrow \text{Result} \]
\[ ++ d03faf(xs, xf, l, lbdcnd, bdxs, bdxf, ys, yf, m, mbdcnd, bdys, bdyf, zs,
++ zf, n, nbdcnd, bdzs, bdzf, lambda, ldimf, mdimf, lwrk, f, ifail) \]
++ solves the Helmholtz equation in Cartesian co-ordinates in
++ three dimensions using the standard seven-point finite difference
++ approximation. This routine is designed to be particularly
++ efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXd03faf}.

Implementation ==> add

\begin{verbatim}
import Lisp
import DoubleFloat
import Any
import Record
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(ThreeDimensionalMatrix DoubleFloat)
import FortranPackage
import Union(fn: FileName, fp: Asp73(PDEF))
import Union(fn: FileName, fp: Asp74(BNDY))
\end{verbatim}

\[ d3edf(ngxArg:Integer,ngyArg:Integer,ldaArg:Integer,\_
maxitArg:Integer,accArg:DoubleFloat,ioutArg:Integer,\_
aArg:Matrix DoubleFloat,rhsArg:Matrix Double Float,\_
ubArg:Matrix DoubleFloat,\_
ifailArg:Integer): \text{Result} ==
\end{verbatim}
\[ (\text{invokeNagman}(\text{NIL$Lisp}_,\_
"d3edf",\_
$Lisp_,)\]
\[ "$us"::S,"u"::S,"numit"::S]$Lisp_,
\[ "$u"::S,"lda"::S]$Lisp_,
\[ "$rhs"::S,"lda"::S]$Lisp_,
\[ "$ub"::S,"ngx"::S,"ngy"::S]$Lisp]$Lisp]$Lisp_\]
\[ "$iout"::S,"numit"::S,"ifail"::S]$Lisp]$Lisp_,
\]
d03ef(xminArg:DoubleFloat, xmaxArg:DoubleFloat, yminArg:DoubleFloat, _
ymaxArg:DoubleFloat, ngxArg:Integer, ngyArg:Integer, _
ldaArg:Integer, schemeArg:String, ifailArg:Integer, _
pdefArg:Union(fn:FileName, fp:Asp73(PDEF)), bndyArg:Union(fn:FileName, _
fp:Asp74(BNDY))): Result ==
pushFortranOutputStack(pdefFilename := aspFilename "pdef")$FOP
if pdefArg case fn
then outputAsFortran(pdefArg.fn)
else outputAsFortran(pdefArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(bndyFilename := aspFilename "bndy")$FOP
if bndyArg case fn
then outputAsFortran(bndyArg.fn)
else outputAsFortran(bndyArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([pdefFilename,bndyFilename]$Lisp, _
"d03ef", _
"bndy"::S,"a"::S,"rhs"::S]$Lisp, _
"a"::S,"lda"::S,7$Lisp]$Lisp, _
["character"::S,"scheme"::S]$Lisp]$Lisp, _
["a"::S,"rhs"::S,"ifail"::S]$Lisp, _
["xminArg::Any, xmaxArg::Any, yminArg::Any, ymaxArg::Any, ngxArg::Any, _
ngyArg::Any, ldaArg::Any, schemeArg::Any, ifailArg::Any ]_ _
@List Any]$Lisp)$Lisp) _
pretend List (Record(key:Symbol,entry:Any))$Result

d03faf(xsArg:DoubleFloat, xfArg:DoubleFloat, lArg:Integer, _
lbdcndArg:Integer, bdxsArg:Matrix DoubleFloat, _
bdfxArg:Matrix DoubleFloat, _
ysArg:DoubleFloat, yfArg:DoubleFloat, mArg:Integer, _
mbdcndArg:Integer, bdysArg:Matrix DoubleFloat, _
bdfyArg:Matrix DoubleFloat, _
zsArg:DoubleFloat, zfArg:DoubleFloat, nArg:Integer, _
nbdcndArg:Integer, bdzsArg:Matrix DoubleFloat, _
bdfzArg:Matrix DoubleFloat, _
lambdaArg:DoubleFloat, ldimfArg:Integer, mdimfArg:Integer, _
lwrkArg:Integer, fArg:ThreeDimensionalMatrix DoubleFloat, _
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"d03faf", _
@List Any]$Lisp)$Lisp) _
pretend List (Record(key:Symbol,entry:Any))$Result
package NAGC02 NagPolynomialRootsPackage

— NagPolynomialRootsPackage.input —

)set break resume
If \( z \) is a zero, then \( f(z) \) can be divided by a factor \((z-z)\): 
\[
f(z) = (z-z) f(z) = (z-z_1) f(z)
\] 
\[
1 1
\]
where \( f(z) \) is a polynomial of degree \( n-1 \). By the Fundamental Theorem of Algebra, a polynomial \( f(z) \) always has a zero, and so the process of dividing out factors \((z-z)\) can be continued until we have a complete factorization of \( f(z) \):
\[
f(z) = a(z-z_0)(z-z_1)...(z-z_n).
\]
Here the complex numbers \( z_0, z_1, ..., z_n \) are the zeros of \( f(z) \); they may not all be distinct, so it is sometimes more convenient to write:
\[
f(z) = a(z-z_0)(z-z_1)...(z-z_k), \quad k \leq n,
\]
with distinct zeros \( z_0, z_1, ..., z_k \) and multiplicities \( m_i \geq 1 \). If \( m_i = 1 \), \( z_i \) is called a single zero, if \( m_i > 1 \), \( z_i \) is called a multiple or repeated zero; a multiple zero is also a zero of the derivative of \( f(z) \).

If the coefficients of \( f(z) \) are all real, then the zeros of \( f(z) \) are either real or else occur as pairs of conjugate complex numbers \( x+iy \) and \( x-iy \). A pair of complex conjugate zeros are the zeros of a quadratic factor of \( f(z) \), \((z^2 + rz + s)\), with real coefficients \( r \) and \( s \).

Mathematicians are accustomed to thinking of polynomials as pleasantly simple functions to work with. However the problem of numerically computing the zeros of an arbitrary polynomial is far from simple. A great variety of algorithms have been proposed, of which a number have been widely used in practice; for a fairly comprehensive survey, see Householder [1]. All general algorithms are iterative. Most converge to one zero at a time; the corresponding factor can then be divided out as in equation (1) above — this process is called deflation or, loosely, dividing out the zero — and the algorithm can be applied again to the
polynomial $f(z)$. A pair of complex conjugate zeros can be divided out together - this corresponds to dividing $f(z)$ by a quadratic factor.

Whatever the theoretical basis of the algorithm, a number of practical problems arise: for a thorough discussion of some of them see Peters and Wilkinson [2] and Wilkinson [3]. The most elementary point is that, even if $z$ is mathematically an exact zero of $f(z)$, because of the fundamental limitations of computer arithmetic the computed value of $f(z)$ will not necessarily be exactly 0.0. In practice there is usually a small region of values of $z$ about the exact zero at which the computed value of $f(z)$ becomes swamped by rounding errors. Moreover in many algorithms this inaccuracy in the computed value of $f(z)$ results in a similar inaccuracy in the computed step from one iterate to the next. This limits the precision with which any zero can be computed. Deflation is another potential cause of trouble, since, in the notation of equation (1), the computed coefficients of $f(z)$ will not be completely accurate, especially if $z$ is not an exact zero of $f(z)$; so the zeros of the computed $f(z)$ will deviate from the zeros of $f(z)$.

A zero is called ill-conditioned if it is sensitive to small changes in the coefficients of the polynomial. An ill-conditioned zero is likewise sensitive to the computational inaccuracies just mentioned. Conversely a zero is called well-conditioned if it is comparatively insensitive to such perturbations. Roughly speaking a zero which is well separated from other zeros is well-conditioned, while zeros which are close together are ill-conditioned, but in talking about 'closeness' the decisive factor is not the absolute distance between neighbouring zeros but their ratio: if the ratio is close to 1 the zeros are ill-conditioned. In particular, multiple zeros are ill-conditioned. A multiple zero is usually split into a cluster of zeros by perturbations in the polynomial or computational inaccuracies.

2.1. References


3. Recommendations on Choice and Use of Routines

3.1. Discussion

Two routines are available: C02AFF for polynomials with complex coefficients and C02AGF for polynomials with real coefficients.

C02AFF and C02AGF both use a variant of Laguerre’s Method due to BT Smith to calculate each zero until the degree of the deflated polynomial is less than 3, whereupon the remaining zeros are obtained using the ’standard’ closed formulae for a quadratic or linear equation.

The accuracy of the roots will depend on how ill-conditioned they are. Peters and Wilkinson [2] describe techniques for estimating the errors in the zeros after they have been computed.

3.2. Index

Zeros of a complex polynomial C02AFF
Zeros of a real polynomial C02AGF

1. Purpose

C02AFF finds all the roots of a complex polynomial equation,
using a variant of Laguerre's Method.

2. Specification

SUBROUTINE C02AFF (A, N, SCALE, Z, W, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION A(2,N+1), Z(2,N), W(4*(N+1))
LOGICAL SCALE

3. Description

The routine attempts to find all the roots of the nth degree complex polynomial equation

\[ P(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \ldots + a_1 z + a_0 = 0. \]

The roots are located using a modified form of Laguerre's Method, originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

\[ L(z) = z_k - z_k = \frac{-n \cdot P(z_k)}{P'(z_k) + \sqrt{H(z_k)}} \]

where \( H(z_k) = (n-1) \cdot n \cdot P''(z_k) \) and \( z_k \) is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at \( z_k \), viz. \(|L(z_k)|\), is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots. The routine generates a sequence of iterates \( z_1, z_2, z_3, \ldots \), such that \(|P(z_k)| < |P(z_{k+1})|\) and ensures that \( z_k + L(z_k) \) 'roughly' lies inside a circular region of radius \(|F_k|\) about \( z_k \) known to contain a zero of \( P(z) \); that is, \(|L(z_k)| \leq |F_k|\), where \( F_k \) denotes the Fejer bound (see Marden [1]) at the point \( z_k \). Following Smith
[2], $F$ is taken to be $\min(B, 1.445n^*R)$, where $B$ is an upper bound for the magnitude of the smallest zero given by

$$B = 1.0001 \times \min(\sqrt{nL(z_k)}, |r|, |a_k/a|),$$

where $r$ is the zero $X$ of smaller magnitude of the quadratic equation

$$2(P''(z)/(2n*(n-1)))X + 2(P'(z)/n)X + P(z) = 0$$

and the Cauchy lower bound $R$ for the smallest zero is computed (using Newton's Method) as the positive root of the polynomial equation

$$|a_0|z^n + |a_1|z^{n-1} + |a_2|z^{n-2} + \ldots + |a_{n-1}|z + |a_n| = 0.$$  

Starting from the origin, successive iterates are generated according to the rule $z_k = z + L(z_k)$ for $k = 1, 2, 3, \ldots$ and $L(z_k)$ is 'adjusted' so that $|P(z_k)| < |P(z_{k-1})|$ and $|L(z_k)| = |F|$. The iterative procedure terminates if $P(z_k)$ is smaller in absolute value than the bound on the rounding error in $P(z_k)$ and the current iterate $z_k$ is taken to be a zero of $P(z)$. The deflated polynomial $P(z) = P(z)/(z - z_k)$ of degree $n-1$ is then formed, and the above procedure is repeated on the deflated polynomial until $n<3$, whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear ($n = 1$) or quadratic ($n = 2$) equation.

To obtain the roots of a quadratic polynomial, C02AHF(*) can be used.

4. References


5. Parameters

1: A(2,N+1) -- DOUBLE PRECISION array Input
   On entry: if A is declared with bounds (2,0:N), then A(1,i) and A(2,i) must contain the real and imaginary parts of a
   \( a_i \) (i.e., the coefficient of \( z^{n-i} \)), for i=0,1,...,n.
   Constraint: A(1,0) /= 0.0 or A(2,0) /= 0.0.

2: N -- INTEGER Input

3: SCALE -- LOGICAL Input
   On entry: indicates whether or not the polynomial is to be scaled. See Section 8 for advice on when it may be preferable to set SCALE = .FALSE. and for a description of the scaling strategy. Suggested value: SCALE = .TRUE..

4: Z(2,N) -- DOUBLE PRECISION array Output
   On exit: the real and imaginary parts of the roots are stored in Z(1,i) and Z(2,i) respectively, for i=1,2,...,n.

5: W(4*(N+1)) -- DOUBLE PRECISION array Workspace

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry $A(1,0) = 0.0$ and $A(2,0) = 0.0$,
or\quad $N < 1$.

IFAIL= 2
The iterative procedure has failed to converge. This error
is very unlikely to occur. If it does, please contact NAG
immediately, as some basic assumption for the arithmetic has
been violated. See also Section 8.

IFAIL= 3
Either overflow or underflow prevents the evaluation of $P(z)$
near some of its zeros. This error is very unlikely to
occur. If it does, please contact NAG immediately. See also
Section 8.

7. Accuracy

All roots are evaluated as accurately as possible, but because of
the inherent nature of the problem complete accuracy cannot be
guaranteed.

8. Further Comments

If SCALE = .TRUE., then a scaling factor for the coefficients is
chosen as a power of the base $B$ of the machine so that the
largest coefficient in magnitude approaches $\text{THRESH} = B^{EMAX-P}$.
Users should note that no scaling is performed if the largest
coefficient in magnitude exceeds $\text{THRESH}$, even if SCALE = .TRUE..
(For definition of $B$, EMAX and P see the Chapter Introduction
X02.)

However, with SCALE = .TRUE., overflow may be encountered when
the input coefficients $a_0, a_1, a_2, \ldots, a_n$ vary widely in magnitude,
\begin{equation}
(4P)
\end{equation}
particularly on those machines for which $B$ overflows. In
such cases, SCALE should be set to .FALSE. and the coefficients
scaled so that the largest coefficient in magnitude does not
exceed $B^{(EMAX-2*P)}$.

Even so, the scaling strategy used in CO2AFF is sometimes
insufficient to avoid overflow and/or underflow conditions. In
such cases, the user is recommended to scale the independent
variable $(z)$ so that the disparity between the largest and
smallest coefficient in magnitude is reduced. That is, use the
routine to locate the zeros of the polynomial $d*P(cz)$ for some
suitable values of $c$ and $d$. For example, if the original
polynomial was \( P(z) = 2i + 2z \), then choosing \( c = 2 \) and \( d = 2 \), for instance, would yield the scaled polynomial \( i + z \), which is well-behaved relative to overflow and underflow and has 10 zeros which are 2 times those of \( P(z) \).

If the routine fails with IFAIL = 2 or 3, then the real and imaginary parts of any roots obtained before the failure occurred are stored in Z in the reverse order in which they were found. Let \( n \) denote the number of roots found before the failure occurred. Then \( Z(1,n) \) and \( Z(2,n) \) contain the real and imaginary parts of the 1st root found, \( Z(1,n-1) \) and \( Z(2,n-1) \) contain the real and imaginary parts of the 2nd root found, ..., \( Z(1,n) \) and \( Z(2,n) \) contain the real and imaginary parts of the \( n \)th root found. After the failure has occurred, the remaining \( 2(n-n) \) elements of Z contain a large negative number (equal to \(-1/(\chi02AMF().\sqrt{2}).\))

9. Example

\[
5 4 3 2
\]

To find the roots of the polynomial \( a_5 z^5 + a_4 z^4 + a_3 z^3 + a_2 z^2 + a_1 z + a_0 = 0 \),

where \( a_0 = (5.0+6.0i) \), \( a_1 = (30.0+20.0i) \), \( a_2 = -(0.2+6.0i) \),

\( a_3 = (50.0+100000.0i) \), \( a_4 = -(2.0-40.0i) \) and \( a_5 = (10.0+1.0i) \).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
not included in the Foundation Library.

1. Purpose

C02AGF finds all the roots of a real polynomial equation, using a variant of Laguerre’s Method.

2. Specification

```
SUBROUTINE C02AGF (A, N, SCALE, Z, W, IFAIL)
  INTEGER N, IFAIL
  DOUBLE PRECISION A(N+1), Z(2,N), W(2*(N+1))
  LOGICAL SCALE
```

3. Description

The routine attempts to find all the roots of the nth degree real polynomial equation

\[ P(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0 = 0. \]

The roots are located using a modified form of Laguerre’s Method, originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

\[
L(z) = z_k - \frac{-n \cdot P(z_k)}{P'(z_k) + \sqrt{H(z_k)}},
\]

where \( H(z) = (n-1) \cdot \left[ (n-1) \cdot (P'(z_k)) - n \cdot P(z_k) \cdot P''(z_k) \right] \), and \( z \) is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at \( z \), viz. \( |L(z)| \), is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots.

The routine generates a sequence of iterates \( z_1, z_2, z_3, \ldots \), such that \( |P(z_{k+1})| < |P(z_k)| \) and ensures that \( z_k + L(z_k) \) ‘roughly’
lies inside a circular region of radius $|F|$ about $z$ known to contain a zero of $P(z)$; that is, $|L(z)| \leq |F|$, where $F$ denotes the Fejer bound (see Marden [1]) at the point $z$. Following Smith [2], $F$ is taken to be $\min(B, 1.445 \times n \times R)$, where $B$ is an upper bound for the magnitude of the smallest zero given by

$$B = 1.0001 \times \min(\sqrt{n} \times L(z), |r|, |a_1/a_n|),$$

where $r$ is the zero $X$ of smaller magnitude of the quadratic equation

$$2 \left(2P''(z)/(2n \times (n-1))X + 2P'(z)/n \right)X + P(z) = 0$$

and the Cauchy lower bound $R$ for the smallest zero is computed (using Newton’s Method) as the positive root of the polynomial equation

$$z^n - z^{n-1} - z^{n-2} - \ldots - z^1 - z^0 = 0.$$ 

Starting from the origin, successive iterates are generated according to the rule $z = z + L(z)$ for $k=1, 2, 3, \ldots$ and $L(z)$ is $\frac{1}{n}$

$$L(z) = \frac{1}{1/n \times L(z), |r|, |a_1/a_n|}.$$ 

The iterative procedure terminates if $P(z)$ is smaller in absolute value than the bound on the rounding error in $P(z)$ and the current iterate $z = z$ is taken to be a zero of $P(z)$ (as is its conjugate $z$ if $z$ is complex). The deflated polynomial

$$P(z) = P(z)/(z-z_k)$$

of degree $n-1$ if $z$ is real

$$P(z) = P(z)/(z-z_k)(z-z_{k+1})$$

of degree $n-2$ if $z$ is complex) is then formed, and the above procedure is repeated on the deflated polynomial until $n < 3$, whereupon the remaining roots are obtained.
via the 'standard' closed formulae for a linear \((n = 1)\) or quadratic \((n = 2)\) equation.

To obtain the roots of a quadratic polynomial, C02AJF(*) can be used.

4. References


5. Parameters

1: A(N+1) -- DOUBLE PRECISION array
   Input
   On entry: if A is declared with bounds (0:N), then A(i)
   must contain a \((i.e., \text{the coefficient of } z^{-i})\), for
   \(i = 0, 1, \ldots, n\). Constraint: A(0) \(!= 0.0\).

2: N -- INTEGER
   Input
   On entry: the degree of the polynomial, \(n\). Constraint: \(N \geq 1\).

3: SCALE -- LOGICAL
   Input
   On entry: indicates whether or not the polynomial is to be scaled. See Section 8 for advice on when it may be preferable to set SCALE = .FALSE. and for a description of the scaling strategy. Suggested value: SCALE = .TRUE..

4: Z(2,N) -- DOUBLE PRECISION array
   Output
   On exit: the real and imaginary parts of the roots are stored in Z(1,i) and Z(2,i) respectively, for \(i = 1, 2, \ldots, n\). Complex conjugate pairs of roots are stored in consecutive pairs of elements of Z; that is, Z(1,i+1) = Z(1,i) and Z(2,i+1) = -Z(2,i).

5: W(2*(N+1)) -- DOUBLE PRECISION array
   Workspace

6: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

<table>
<thead>
<tr>
<th>IFAIL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On entry A(0) = 0.0, or N &lt; 1.</td>
</tr>
<tr>
<td>2</td>
<td>The iterative procedure has failed to converge. This error is very unlikely to occur. If it does, please contact NAG immediately, as some basic assumption for the arithmetic has been violated. See also Section 8.</td>
</tr>
<tr>
<td>3</td>
<td>Either overflow or underflow prevents the evaluation of P(z) near some of its zeros. This error is very unlikely to occur. If it does, please contact NAG immediately. See also Section 8.</td>
</tr>
</tbody>
</table>

7. Accuracy

All roots are evaluated as accurately as possible, but because of the inherent nature of the problem complete accuracy cannot be guaranteed.

8. Further Comments

If SCALE = .TRUE., then a scaling factor for the coefficients is chosen as a power of the base B of the machine so that the largest coefficient in magnitude approaches THRESH = B ** (EMAX-P).

Users should note that no scaling is performed if the largest coefficient in magnitude exceeds THRESH, even if SCALE = .TRUE. (For definition of B, EMAX and P see the Chapter Introduction X02.)

However, with SCALE = .TRUE., overflow may be encountered when the input coefficients a_0, a_1, a_2, ..., a_n vary widely in magnitude, particularly on those machines for which B overflows. In
such cases, SCALE should be set to .FALSE. and the coefficients scaled so that the largest coefficient in magnitude does not exceed $B$.

Even so, the scaling strategy used in C02AGF is sometimes insufficient to avoid overflow and/or underflow conditions. In such cases, the user is recommended to scale the independent variable ($z$) so that the disparity between the largest and smallest coefficient in magnitude is reduced. That is, use the routine to locate the zeros of the polynomial $d\times P(cz)$ for some suitable values of $c$ and $d$. For example, if the original polynomial was $P(z)=2 +2z\ , \text{ then choosing } c=2 \text{ and } d=2\ , \text{ for instance, would yield the scaled polynomial } 1+z\ , \text{ which is well-behaved relative to overflow and underflow and has 10 zeros which are 2 times those of } P(z)$. 

If the routine fails with $IFAIL = 2$ or $3$, then the real and imaginary parts of any roots obtained before the failure occurred are stored in $Z$ in the reverse order in which they were found. Let $n$ denote the number of roots found before the failure occurred. Then $Z(1,n)$ and $Z(2,n)$ contain the real and imaginary parts of the $1$st root found, $Z(1,n-1)$ and $Z(2,n-1)$ contain the real and imaginary parts of the $2$nd root found, ..., $Z(1,n)$ and $Z(2,n)$ contain the real and imaginary parts of the $n$th root found. After the failure has occurred, the remaining $2\times(n-n)$ elements of $Z$ contain a large negative number (equal to $-1/(\text{X02AMF}()\cdot\sqrt{2})$).

9. Example

To find the roots of the 5th degree polynomial

$$z^5 +2z^4 +3z^3 +4z^2 +5z+6=0.$$ 

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagPolynomialRootsPackage (NAGC02)

Exports:
  c02aff  c02agf

— package NAGC02 NagPolynomialRootsPackage —

)abbrev package NAGC02 NagPolynomialRootsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:27 1994
++ Description:
++ This package uses the NAG Library to compute the zeros of a
++ polynomial with real or complex coefficients.

NagPolynomialRootsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports == with
  c02aff : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
    ++ finds all the roots of a complex polynomial equation,
    ++ using a variant of Laguerre's Method.
    ++ See \downlink{Manual Page}{manpageXXc02aff}.
  c02agf : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
    ++ finds all the roots of a real polynomial equation, using a
    ++ variant of Laguerre's Method.
    ++ See \downlink{Manual Page}{manpageXXc02agf}.

Implementation == add

import Lisp
import DoubleFloat
import Matrix DoubleFloat
import Any
import Record
import Integer
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(Boolean)

c02aff(aArg:Matrix DoubleFloat,nArg:Integer,scaleArg:Boolean,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c02aff",_
["z":S,"w":S]$Lisp_,
[["double":S,"a":S,2$Lisp,["+":S,"n":S,1$Lisp]$Lisp]$Lisp_,
["z":S,2$Lisp,"n":S]$Lisp,["w":S,["+":S,
["+":S,"n":S,1$Lisp]$Lisp,4$Lisp]$Lisp]$Lisp]$Lisp_,
["integer":S,"n":S,"ifail":S]$Lisp_,
["logical":S,"scale":S]$Lisp$_
]$Lisp_,
["z":S,"ifail":S]$Lisp_,
([nArg::Any,scaleArg::Any,ifailArg::Any,aArg::Any ])_
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c02agf(aArg:Matrix DoubleFloat,nArg:Integer,scaleArg:Boolean,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c02agf",_
["z":S,"w":S]$Lisp_,
[["double":S,"a":S,["+":S,"n":S,1$Lisp]$Lisp]$Lisp_,
["z":S,2$Lisp,"n":S]$Lisp,["w":S,["+":S,
["+":S,"n":S,1$Lisp]$Lisp,2$Lisp]$Lisp]$Lisp]$Lisp_,
["integer":S,"n":S,"ifail":S]$Lisp_,
["logical":S,"scale":S]$Lisp$_
]$Lisp_,
["z":S,"ifail":S]$Lisp_,
([nArg::Any,scaleArg::Any,ifailArg::Any,aArg::Any ])_
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

— NAGC02.dotabb —
package NAGC05 NagRootFindingPackage

--- NagRootFindingPackage.input ---

)set break resume
)sys rm -f NagRootFindingPackage.output
)spool NagRootFindingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagRootFindingPackage
--E 1

)spool
)lisp (bye)

--- NagRootFindingPackage.help ---

This package uses the NAG Library to calculate real zeros of continuous
real functions of one or more variables. (Complex equations must be
expressed in terms of the equivalent larger system of real equations.)

C05(3NAG) Foundation Library (12/10/92) C05(3NAG)

C05 -- Roots of One or More Transcendental Equations
       Introduction -- C05

Chapter C05
Roots of One or More Transcendental Equations

1. Scope of the Chapter

This chapter is concerned with the calculation of real zeros of
continuous real functions of one or more variables. (Complex
equations must be expressed in terms of the equivalent larger
system of real equations.)
2. Background to the Problems

The chapter divides naturally into two parts.

2.1. A Single Equation

The first deals with the real zeros of a real function of a single variable \( f(x) \).

At present, there is only one routine with a simple calling sequence. This routine assumes that the user can determine an initial interval \([a, b]\) within which the desired zero lies, that is \( f(a) \cdot f(b) < 0 \), and outside which all other zeros lie. The routine then systematically subdivides the interval to produce a final interval containing the zero. This final interval has a length bounded by the user’s specified error requirements; the end of the interval where the function has smallest magnitude is returned as the zero. This routine is guaranteed to converge to a simple zero of the function. (Here we define a simple zero as a zero corresponding to a sign-change of the function.) The algorithm used is due to Bus and Dekker.

2.2. Systems of Equations

The routines in the second part of this chapter are designed to solve a set of nonlinear equations in \( n \) unknowns

\[
T \begin{align*}
  f(x) &= 0, \quad i=1,2,...,n, \quad x = (x_1, x_2, ..., x_n) \\
\end{align*}
\]

where \( T \) stands for transpose.

It is assumed that the functions are continuous and differentiable so that the matrix of first partial derivatives of the functions, the Jacobian matrix \( J(x) = \frac{df}{dx} \) evaluated at

\[
\begin{bmatrix}
  \frac{\partial f_i}{\partial x_j}
\end{bmatrix}_{i,j=1}^n
\]

the point \( x \), exists, though it may not be possible to calculate it directly.

The functions \( f \) must be independent, otherwise there will be an infinity of solutions and the methods will fail. However, even when the functions are independent the solutions may not be unique. Since the methods are iterative, an initial guess at the solution has to be supplied, and the solution located will usually be the one closest to this initial guess.
2.3. References


3. Recommendations on Choice and Use of Routines

3.1. Zeros of Functions of One Variable

There is only one routine (C05ADF) for solving a single nonlinear equation. This routine is designed for solving problems where the function \( f(x) \) whose zero is to be calculated, can be coded as a user-supplied routine.

C05ADF may only be used when the user can supply an interval \([a,b]\) containing the zero, that is \( f(a)f(b)<0 \).

3.2. Solution of Sets of Nonlinear Equations

The solution of a set of nonlinear equations

\[
\begin{align*}
  f(x_i,x_2,...,x_n) &= 0, \quad i=1,2,...,n \\
  \end{align*}
\]

(2)

can be regarded as a special case of the problem of finding a minimum of a sum of squares

\[
\begin{align*}
  s(x) &= \sum_{i=1}^{m} \left[ f(x_i,x,...,x_n) \right]^2 \quad \text{for} \quad m\geq n. \\
  \end{align*}
\]

(3)

So the routines in Chapter E04 of the Library are relevant as well as the special nonlinear equations routines.

There are two routines (C05NBF and C05PBF) for solving a set of nonlinear equations. These routines require the \( f \) (and possibly their derivatives) to be calculated in user-supplied routines.
These should be set up carefully so the Library routines can work as efficiently as possible.

The main decision which has to be made by the user is whether to supply the derivatives. It is advisable to do so if possible, since the results obtained by algorithms which use derivatives are generally more reliable than those obtained by algorithms which do not use derivatives.

C05PBF requires the user to provide the derivatives, whilst C05NBF does not. C05NBF and C05PBF are easy-to-use routines. A routine, C05ZAF, is provided for use in conjunction with C05PBF to check the user-provided derivatives for consistency with the functions themselves. The user is strongly advised to make use of this routine whenever C05PBF is used.

Firstly, the calculation of the functions and their derivatives should be ordered so that cancellation errors are avoided. This is particularly important in a routine that uses these quantities to build up estimates of higher derivatives.

Secondly, scaling of the variables has a considerable effect on the efficiency of a routine. The problem should be designed so that the elements of x are of similar magnitude. The same comment applies to the functions, all the fi should be of comparable size.

The accuracy is usually determined by the accuracy parameters of the routines, but the following points may be useful:

(i) Greater accuracy in the solution may be requested by choosing smaller input values for the accuracy parameters. However, if unreasonable accuracy is demanded, rounding errors may become important and cause a failure.

(ii) Some idea of the accuracies of the xi may be obtained by monitoring the progress of the routine to see how many figures remain unchanged during the last few iterations.

(iii) An approximation to the error in the solution x, given by e where e is the solution to the set of linear equations

\[ J(x)e = -f(x) \]
where \( f(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \) (see Chapter F04).

(iv) If the functions \( f_i(x) \) are changed by small amounts \( \epsilon_i \), for \( i=1,2,\ldots,n \), then the corresponding change in the solution \( x \) is given approximately by \( \sigma \), where \( \sigma \) is the solution of the set of linear equations

\[
J(x)(\sigma) = -\epsilon
\]

(see Chapter F04).

Thus one can estimate the sensitivity of \( x \) to any uncertainties in the specification of \( f_i(x) \), for \( i=1,2,\ldots,n \).

3.3. Index

Zeros of functions of one variable:
- Bus and Dekker algorithm: C05ADF

Zeros of functions of several variables:
- easy-to-use: C05NBF
- easy-to-use, derivatives required: C05PBF

Checking Routine:
- Checks user-supplied Jacobian: C05ZAF

C05 -- Roots of One or More Transcendental Equations

Chapter C05

Roots of One or More Transcendental Equations

C05ADF Zero of continuous function in given interval, Bus and Dekker algorithm

C05NBF Solution of system of nonlinear equations using function values only

C05PBF Solution of system of nonlinear equations using 1st derivatives

C05ZAF Check user's routine for calculating 1st derivatives
1. Purpose

C05ADF locates a zero of a continuous function in a given interval by a combination of the methods of linear interpolation, extrapolation and bisection.

2. Specification

```fortran
SUBROUTINE C05ADF (A, B, EPS, ETA, F, X, IFAIL)
    INTEGER IFAIL
    DOUBLE PRECISION A, B, EPS, ETA, F, X
    EXTERNAL F
```

3. Description

The routine attempts to obtain an approximation to a simple zero of the function \( f(x) \) given an initial interval \([a, b]\) such that \( f(a) \cdot f(b) \leq 0 \). The zero is found by calls to C05AZF(*) whose specification should be consulted for details of the method used.

The approximation \( x \) to the zero (\( \alpha \)) is determined so that one or both of the following criteria are satisfied:

(i) \( |x-\alpha| < \text{EPS} \),

(ii) \( |f(x)| < \text{ETA} \).

4. References

None.

5. Parameters

1:  A -- DOUBLE PRECISION \hspace{1cm} \text{Input}
    On entry: the lower bound of the interval, \( a \).

2:  B -- DOUBLE PRECISION \hspace{1cm} \text{Input}
    On entry: the upper bound of the interval, \( b \). Constraint: \( B \neq A \).

3:  EPS -- DOUBLE PRECISION \hspace{1cm} \text{Input}
    On entry: the absolute tolerance to which the zero is
required (see Section 3). Constraint: EPS > 0.0.

4: ETA -- DOUBLE PRECISION  
   Input  
   On entry: a value such that if |f(x)|<ETA, x is accepted as 
   the zero. ETA may be specified as 0.0 (see Section 7).

5: F -- DOUBLE PRECISION FUNCTION, supplied by the user.  
   External Procedure  
   F must evaluate the function f whose zero is to be 
   determined.

   Its specification is:

   DOUBLE PRECISION FUNCTION F (XX)  
   DOUBLE PRECISION XX  

   1: XX -- DOUBLE PRECISION  
      Input  
      On entry: the point at which the function must be 
      evaluated.

   F must be declared as EXTERNAL in the (sub)program from 
   which C05ADF is called. Parameters denoted as Input 
   must not be changed by this procedure.

6: X -- DOUBLE PRECISION  
   Output  
   On exit: the approximation to the zero.

7: IFAIL -- INTEGER  
   Input/Output  
   Before entry, IFAIL must be assigned a value. For users not 
   familiar with this parameter (described in the Essential 
   Introduction) the recommended value is 0.

   Unless the routine detects an error (see Section 6), IFAIL 
   contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1  
On entry EPS <= 0.0,  
or A = B,  
or F(A)*F(B)>0.0.

IFAIL= 2  
Too much accuracy has been requested in the computation,  
that is, EPS is too small for the computer being used. The 
final value of X is an accurate approximation to the zero.
CHAPTER 15. CHAPTER N

IFAIL= 3
A change in sign of f(x) has been determined as occurring
near the point defined by the final value of X. However,
there is some evidence that this sign-change corresponds to
a pole of f(x).

IFAIL= 4
Indicates that a serious error has occurred in C05AZF(*).
Check all routine calls. Seek expert help.

7. Accuracy

This depends on the value of EPS and ETA. If full machine
accuracy is required, they may be set very small, resulting in an
error exit with IFAIL = 2, although this may involve more
iterations than a lesser accuracy. The user is recommended to set
ETA = 0.0 and to use EPS to control the accuracy, unless he has
considerable knowledge of the size of f(x) for values of x near
the zero.

8. Further Comments

The time taken by the routine depends primarily on the time spent
evaluating F (see Section 5).

If it is important to determine an interval of length less than
EPS containing the zero, or if the function F is expensive to
evaluate and the number of calls to F is to be restricted, then
use of C05AZF(*) is recommended. Use of C05AZF(*) is also
recommended when the structure of the problem to be solved does
not permit a simple function F to be written: the reverse
communication facilities of C05AZF(*) are more flexible than the
direct communication of F required by C05ADF.

9. Example

The example program below calculates the zero of $e^{-x}$ within the
interval [0,1] to approximately 5 decimal places.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C05NBF is an easy-to-use routine to find a solution of a system of nonlinear equations by a modification of the Powell hybrid method.

2. Specification

SUBROUTINE C05NBF (FCN, N, X, FVEC, XTOL, WA, LWA, IFAIL)
INTEGER N, LWA, IFAIL
DOUBLE PRECISION X(N), FVEC(N), XTOL, WA(LWA)
EXTERNAL FCN

3. Description

The system of equations is defined as:

\[ f(x_1, x_2, ..., x_n) = 0, \text{ for } i = 1, 2, ..., n. \]

C05NBF is based upon the MINPACK routine HYBRD1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is approximated by forward differences, but these are not used again until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].

4. References


5. Parameters

1: FCN -- SUBROUTINE, supplied by the user.
External Procedure
FCN must return the values of the functions $f_i$ at a point $x$.

Its specification is:

```fortran
SUBROUTINE FCN (N, X, FVEC, IFLAG)
INTEGER N, IFLAG
DOUBLE PRECISION X(N), FVEC(N)

1: N -- INTEGER  
   On entry: the number of equations, n.

2: X(N) -- DOUBLE PRECISION array  
   On entry: the components of the point $x$ at which the functions must be evaluated.

3: FVEC(N) -- DOUBLE PRECISION array  
   On exit: the function values $f_i(x)$ (unless IFLAG is set to a negative value by FCN).

4: IFLAG -- INTEGER  
   On entry: IFLAG > 0. On exit: in general, IFLAG should not be reset by FCN. If, however, the user wishes to terminate execution (perhaps because some illegal point $X$ has been reached), then IFLAG should be set to a negative integer. This value will be returned through IFAIL.

FCN must be declared as EXTERNAL in the (sub)program from which C05NBF is called. Parameters denoted as Input must not be changed by this procedure.
```

2: N -- INTEGER  
   On entry: the number of equations, n. Constraint: $N > 0$.

3: X(N) -- DOUBLE PRECISION array  
   On entry: an initial guess at the solution vector. On exit: the final estimate of the solution vector.

4: FVEC(N) -- DOUBLE PRECISION array  
   On exit: the function values at the final point, $X$.

5: XTOL -- DOUBLE PRECISION  
   On entry: the accuracy in $X$ to which the solution is required. Suggested value: the square root of the machine precision. Constraint: $XTOL \geq 0.0$.

6: WA(LWA) -- DOUBLE PRECISION array  
   Workspace
7: LWA -- INTEGER
    On entry: the dimension of the array WA. Constraint:
    LWA >= N*(3*N+13)/2.

8: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL< 0
    The user has set IFLAG negative in FCN. The value of IFAIL
    will be the same as the user’s setting of IFLAG.

IFAIL= 1
    On entry N <= 0,
    or XTOL < 0.0,
    or LWA < N*(3*N+13)/2.

IFAIL= 2
    There have been at least 200*(N+1) evaluations of FCN.
    Consider restarting the calculation from the final point
    held in X.

IFAIL= 3
    No further improvement in the approximate solution X is
    possible; XTOL is too small.

IFAIL= 4
    The iteration is not making good progress. This failure exit
    may indicate that the system does not have a zero, or that
    the solution is very close to the origin (see Section 7).
    Otherwise, rerunning COSNBF from a different starting point
    may avoid the region of difficulty.

7. Accuracy

If x is the true solution, COSNBF tries to ensure that
\[ ||x - x|| \leq XTOL \cdot ||x||. \]

If this condition is satisfied with XTOL = 10, then the larger components of \( x \) have \( k \) significant decimal digits. There is a danger that the smaller components of \( x \) may have large relative errors, but the fast rate of convergence of \( \text{C05NBF} \) usually avoids this possibility.

If XTOL is less than machine precision, and the above test is satisfied with the machine precision in place of XTOL, then the routine exits with IFAIL = 3.

Note: this convergence test is based purely on relative error, and may not indicate convergence if the solution is very close to the origin.

The test assumes that the functions are reasonably well behaved. If this condition is not satisfied, then \( \text{C05NBF} \) may incorrectly indicate convergence. The validity of the answer can be checked, for example, by rerunning \( \text{C05NBF} \) with a tighter tolerance.

8. Further Comments

The time required by \( \text{C05NBF} \) to solve a given problem depends on \( n \), the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by \( \text{C05NBF} \) to process each call of FCN is about \( 11.5 \cdot n \). Unless FCN can be evaluated quickly, the timing of \( \text{C05NBF} \) will be strongly influenced by the time spent in FCN.

Ideally the problem should be scaled so that at the solution the function values are of comparable magnitude.

9. Example

To determine the values \( x_1, \ldots, x_9 \) which satisfy the tridiagonal equations:

\[
\begin{align*}
(3-2x)x_{i-1} - 2x &= -1, \quad i = 1, 2, \\
-x_{i-1} + (3-2x)x_i - 2x &= -1, \quad i = 2, 3, \ldots, 8, \\
-x_{i-1} + (3-2x)x_i &= -1.
\end{align*}
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose

C05PBF is an easy-to-use routine to find a solution of a system of nonlinear equations by a modification of the Powell hybrid method. The user must provide the Jacobian.

2. Specification

SUBROUTINE C05PBF (FCN, N, X, FVEC, FJAC, LDFJAC, XTOL, WA, LWA, IFAIL)
INTEGER N, LDFJAC, LWA, IFAIL
DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N), XTOL, WA
EXTERNAL FCN

3. Description

The system of equations is defined as:

\[ f(x_i, x_1, \ldots, x_n) = 0, \quad i = 1, 2, \ldots, n. \]

C05PBF is based upon the MINPACK routine HYBRJ1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is calculated, but it is not recalculated until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].
4. References


5. Parameters

1: FCN -- SUBROUTINE, supplied by the user. External Procedure

Depending upon the value of IFLAG, FCN must either return
the values of the functions $f_i$ at a point $x$ or return the
Jacobian at $x$.

Its specification is:

```fortran
SUBROUTINE FCN (N, X, FVEC, FJAC, LDFJAC, IFLAG)
INTEGER N, LDFJAC, IFLAG
DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N)
```

1: N -- INTEGER Input
On entry: the number of equations, $n$.

2: X(N) -- DOUBLE PRECISION array Input
On entry: the components of the point $x$ at which the
functions or the Jacobian must be evaluated.

3: FVEC(N) -- DOUBLE PRECISION array Output
On exit: if IFLAG = 1 on entry, FVEC must contain the
function values $f_i(x)$ (unless IFLAG is set to a
negative value by FCN). If IFLAG = 2 on entry, FVEC
must not be changed.

4: FJAC(LDFJAC,N) -- DOUBLE PRECISION array Output
On exit: if IFLAG = 2 on entry, FJAC(i,j) must contain
the value of $\frac{df_i}{dx_j}$ at the point $x$, for $i,j=1,2,...,n$
(unless IFLAG is set to a negative value by FCN).

If IFLAG = 1 on entry, FJAC must not be changed.
5: LDFJAC -- INTEGER  
   On entry: the first dimension of FJAC.

6: IFLAG -- INTEGER  
   On entry: IFLAG = 1 or 2: 
     if IFLAG = 1, FVEC is to be updated;
     if IFLAG = 2, FJAC is to be updated.
   On exit: in general, IFLAG should not be reset by FCN.
   If, however, the user wishes to terminate execution 
   (perhaps because some illegal point x has been reached)
   then IFLAG should be set to a negative integer. This 
   value will be returned through IFAIL.

FCN must be declared as EXTERNAL in the (sub)program 
from which C05PBF is called. Parameters denoted as 
Input must not be changed by this procedure.

2: N -- INTEGER  
   On entry: the number of equations, n. Constraint: N > 0.

3: X(N) -- DOUBLE PRECISION array  
   On entry: an initial guess at the solution vector. On 
   exit: the final estimate of the solution vector.

4: FVEC(N) -- DOUBLE PRECISION array  
   On exit: the function values at the final point, X.

5: FJAC(LDFJAC,N) -- DOUBLE PRECISION array  
   On exit: the orthogonal matrix Q produced by the QR 
   factorization of the final approximate Jacobian.

6: LDFJAC -- INTEGER  
   On entry: the first dimension of the array FJAC as declared in the 
   (sub)program from which C05PBF is called. 
   Constraint: LDFJAC >= N.

7: XTOL -- DOUBLE PRECISION  
   On entry: the accuracy in X to which the solution is 
   required. Suggested value: the square root of the machine 
   precision. Constraint: XTOL >= 0.0.

8: WA(LWA) -- DOUBLE PRECISION array  
   Workspace

9: LWA -- INTEGER  
   On entry: the dimension of the array WA. Constraint: 
   LWA>=N*(N+13)/2.

10: IFAIL -- INTEGER  
    On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL< 0
A negative value of IFAIL indicates an exit from C05PBF because the user has set IFLAG negative in FCN. The value of IFAIL will be the same as the user's setting of IFLAG.

IFAIL= 1
On entry N <= 0,
or LDFJAC < N,
or XTOL < 0.0,
or LWA< N*(N+13)/2.

IFAIL= 2
There have been 100*(N+1) evaluations of the functions. Consider restarting the calculation from the final point held in X.

IFAIL= 3
No further improvement in the approximate solution X is possible; XTOL is too small.

IFAIL= 4
The iteration is not making good progress. This failure exit may indicate that the system does not have a zero or that the solution is very close to the origin (see Section 7). Otherwise, rerunning C05PBF from a different starting point may avoid the region of difficulty.

7. Accuracy

If x is the true solution, C05PBF tries to ensure that

\[ ||x - x|| \leq XTOL*||x||. \]
If this condition is satisfied with XTOL=10^−k, then the larger components of x have k significant decimal digits. There is a danger that the smaller components of x may have large relative errors, but the fast rate of convergence of C05PBF usually avoids the possibility.

If XTOL is less than machine precision and the above test is satisfied with the machine precision in place of XTOL, then the routine exits with IFAIL = 3.

Note: this convergence test is based purely on relative error, and may not indicate convergence if the solution is very close to the origin.

The test assumes that the functions and Jacobian are coded consistently and that the functions are reasonably well behaved. If these conditions are not satisfied then C05PBF may incorrectly indicate convergence. The coding of the Jacobian can be checked using C05ZAF. If the Jacobian is coded correctly, then the validity of the answer can be checked by rerunning C05PBF with a tighter tolerance.

8. Further Comments

The time required by C05PBF to solve a given problem depends on n, the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by C05PBF is about 11.5*n to process each evaluation of the functions and about 1.3*n to process each evaluation of the Jacobian. Unless FCN can be evaluated quickly, the timing of C05PBF will be strongly influenced by the time spent in FCN.

Ideally the problem should be scaled so that, at the solution, the function values are of comparable magnitude.

9. Example

To determine the values x_1, ..., x_9 which satisfy the tridiagonal equations:

\[(3-2x_i)x_i -2x_i =-1, \quad i=1,2 \quad (i=2,3,\ldots,8).\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

NagRootFindingPackage (NAGC05)

Exports:
c05adf  c05nbf  c05pbf

— package NAGC05 NagRootFindingPackage —

NagRootFindingPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
c05adf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_,
       Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
++ c05adf(a,b,eps,eta,ifail,f)
++ locates a zero of a continuous function in a given
++ interval by a combination of the methods of linear interpolation,
++ extrapolation and bisection.
++ See \downlink{Manual Page}{manpageXXc05adf}.
c05nbf : (Integer,Integer,Matrix DoubleFloat,DoubleFloat,_,
       Integer,Union(fn:FileName,fp:Asp6(FCN))) -> Result
++ c05nbf(n,lwa,x,xtol,ifail,fcn)
++ is an easy-to-use routine to find a solution of a system
++ of nonlinear equations by a modification of the Powell hybrid
++ method.
++ See \downlink{Manual Page}{manpageXXc05nbf}.
c05pbf : (Integer,Integer,Integer,Matrix DoubleFloat,_,
       DoubleFloat,Integer,Union(fn:FileName,fp:Asp35(FCN))) -> Result
++ c05pbf(n,ldfjac,lwa,x,xtol,ifail,fcn)
++ is an easy-to-use routine to find a solution of a system
++ of nonlinear equations by a modification of the Powell hybrid
++ method. The user must provide the Jacobian.
++ See \downlink{Manual Page}{manpageXXc05pbf}.
Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp1(F))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix Double Float)
import AnyFunctions1(Integer)

c05adf(aArg:DoubleFloat,bArg:DoubleFloat,epsArg:DoubleFloat,_,
       etaArg:DoubleFloat,ifailArg:Integer,_,
       fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
pushForTranOutputStack(fFilename := aspFilename "f")$FOP
if fArg case fn
  then outputAsFortran(fArg.fn)
else outputAsFortran(fArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fFilename]$Lisp,_,
"c05adf",_,
"a":S,"b":S,"eps":S,"eta":S,"x":S_,
"ifail":S,"f":S])$Lisp,_,
"x":S,"f":S]$Lisp,
CHAPTER 15. CHAPTER N

```lisp
["x"::S,"ifail"::S]$$Lisp_ ,
[[[aArg::Any,bArg::Any,epsArg::Any,etaArg::Any,ifailArg::Any ]]
 @List Any]$$Lisp)$$Lisp)$$Lisp)
pretend List (Record(key:Symbol,entry:Any))$$Lisp

pretend List (Record(key:Symbol,entry:Any))$$Lisp

[["integer"::S,"n"::S]$$Lisp_ ,[["double"::S,
[["double"::S,"fvec"::S,"n"::S]$$Lisp_ ,
["fvec"::S,"wa"::S,"fcn"::S]$$Lisp_ ,
[[[aArg::Any,bArg::Any,epsArg::Any,etaArg::Any,ifailArg::Any ]]
 @List Any]$$Lisp)$$Lisp)
pretend List (Record(key:Symbol,entry:Any))$$Lisp

pretend List (Record(key:Symbol,entry:Any))$$Lisp

["fvec"::S,"wa"::S,"fcn"::S]$$Lisp_ ,
[[[aArg::Any,bArg::Any,epsArg::Any,etaArg::Any,ifailArg::Any ]]
 @List Any]$$Lisp)$$Lisp)
pretend List (Record(key:Symbol,entry:Any))$$Lisp

pretend List (Record(key:Symbol,entry:Any))$$Lisp
```

c05nbf(nArg::Integer,lwaArg::Integer,xArg:Matrix DoubleFloat,_
xtolArg::DoubleFloat,ifailArg::Integer,_
fcnArg::Union(fn:FileName,fp:Asp6(FCN))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP

\[(\text{invokeNagman}([\text{fcnFilename}]$$Lisp_ ,
"c05nbf",_
["fvec"::S,"wa"::S,"fcn"::S]$$Lisp_ ,
[[[\text{aArg::Any},\text{bArg::Any},\text{epsArg::Any},\text{etaArg::Any},\text{ifailArg::Any }]
 @List Any]$$Lisp)$$Lisp)$$Lisp)
pretend List (Record(key:Symbol,entry:Any))$$Lisp

pretend List (Record(key:Symbol,entry:Any))$$Lisp

c05pbf(nArg::Integer,ldfjacArg::Integer,lwaArg::Integer,_
xArg:Matrix DoubleFloat,xtolArg::DoubleFloat,ifailArg::Integer,_
fcnArg::Union(fn:FileName,fp:Asp35(FCN))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP

\[(\text{invokeNagman}([\text{fcnFilename}]$$Lisp_ ,
"c05pbf",_
["fvec"::S,"fjac"::S,"wa"::S,"fcn"::S]$$Lisp_ ,
[[[\text{aArg::Any},\text{ldfjacArg::Any},\text{xtolArg::Any},\text{ifailArg::Any},\text{xArg::Any }]
 @List Any]$$Lisp)$$Lisp)$$Lisp)
pretend List (Record(key:Symbol,entry:Any))$$Lisp

pretend List (Record(key:Symbol,entry:Any))$$Lisp
```
This package uses the NAG Library to calculate the discrete Fourier transform of a sequence of real or complex data values, and applies it to calculate convolutions and correlations.
1. Scope of the Chapter

This chapter is concerned with calculating the discrete Fourier transform of a sequence of real or complex data values, and applying it to calculate convolutions and correlations.

2. Background to the Problems

2.1. Discrete Fourier Transforms

2.1.1. Complex transforms

Most of the routines in this chapter calculate the finite discrete Fourier transform (DFT) of a sequence of \( n \) complex numbers \( z_j \), for \( j=0,1,...,n-1 \). The transform is defined by:

\[
\hat{z}_k = \sum_{j=0}^{n-1} z_j e^{-i \frac{2\pi jk}{n}} \tag{1}
\]

for \( k=0,1,...,n-1 \). Note that equation (1) makes sense for all integral \( k \) and with this extension \( z \) is periodic with period \( n \), i.e. \( z_k = z_{k+n} \) and in particular \( z_k = z_{k-n} \).

If we write \( z = x + iy \) and \( z = a + ib \), then the definition of \( z \) may be written in terms of sines and cosines as:

\[
a_k = \sum_{j=0}^{n-1} (x \cos \left( \frac{2\pi jk}{n} \right) + y \sin \left( \frac{2\pi jk}{n} \right)) \tag{2}
\]
\[
\begin{align*}
\sum_{j=0}^{n-1} (2\pi j k) (2\pi j k) \\
b = \sum_{k=1}^{\sqrt{n}} (y \cos(2\pi j k) - x \sin(2\pi j k)) \\
\sqrt{n} \sum_{j=0}^{n-1} (j/n j/n)
\end{align*}
\]

The original data values \( z \) may conversely be recovered from the transform \( z \) by an inverse discrete Fourier transform:

\[
\begin{align*}
\sum_{j=0}^{n-1} \frac{1}{\sqrt{n}} (2\pi j k) \\
z = \sum_{k=0}^{\sqrt{n}} z \exp(i2\pi j k) (2)
\end{align*}
\]

for \( j=0,1,\ldots,n-1 \). If we take the complex conjugate of (2), we find that the sequence \( z \) is the DFT of the sequence \( z \). Hence the inverse DFT of the sequence \( z \) may be obtained by: taking the complex conjugates of the \( z \); performing a DFT; and taking the complex conjugates of the result.

Notes: definitions of the discrete Fourier transform vary. Sometimes (2) is used as the definition of the DFT, and (1) as the definition of the inverse. Also the scale-factor of \( 1/\sqrt{n} \) may be omitted in the definition of the DFT, and replaced by \( 1/n \) in the definition of the inverse.

2.1.2. Real transforms

If the original sequence is purely real valued, i.e. \( z = x \), then

\[
\begin{align*}
\sum_{j=0}^{n-1} 2\pi j k \\
z = a + ib = \sum_{k=0}^{\sqrt{n}} x \exp(-i2\pi j k) \\
k k k \sum_{j=0}^{n-1} (j/n)
\end{align*}
\]
and $z$ is the complex conjugate of $z$. Thus the DFT of a real sequence is a particular type of complex sequence, called a Hermitian sequence, or half-complex or conjugate symmetric with the properties:

$$a_k = a^{-k}, \quad b_k = -b^{-k}, \quad b_0 = 0, \quad b_j = 0$$

Thus a Hermitian sequence of $n$ complex data values can be represented by only $n$, rather than $2n$, independent real values. This can obviously lead to economies in storage, the following scheme being used in this chapter: the real parts $a_k$ for $0 \leq k \leq n/2$ are stored in normal order in the first $n/2+1$ locations of an array $X$ of length $n$; the corresponding non-zero imaginary parts are stored in reverse order in the remaining locations of $X$. In other words, if $X$ is declared with bounds $(0:n-1)$ in the user's (sub)program, the real and imaginary parts of $z$ are stored as follows:

<table>
<thead>
<tr>
<th>$X(k)$</th>
<th>$a_k$</th>
<th>$b_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X(0)$</td>
<td>$a_0$</td>
<td>0</td>
</tr>
<tr>
<td>$X(1)$</td>
<td>$a_1$</td>
<td>1</td>
</tr>
<tr>
<td>$X(2)$</td>
<td>$a_2$</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$X(s)$</td>
<td>$a_s$</td>
<td>$b_s$</td>
</tr>
<tr>
<td>$X(s+1)$</td>
<td>$b_{s-1}$</td>
<td>$b_{s-2}$</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
X(n-2) & \quad b \quad b \\
& \quad 2 \quad 2 \\
X(n-1) & \quad b \quad b \\
& \quad 1 \quad 1 \\
\end{align*}
\]

\[
\left( \begin{array}{c}
\frac{n}{2} - 1 \\
1
\end{array} \right) \\
\left( \begin{array}{c}
(2\pi)jk \\
(2\pi)jk
\end{array} \right)
\]

Hence
\[
x = \frac{1}{\sqrt{n}} \sum_{j=0}^{n/2-1} (a + 2 > (a \cos(\frac{-2\pi j k}{n}) - b \sin(\frac{-2\pi j k}{n})) + a)
\]

where \( a = 0 \) if \( n \) is odd.

\[\frac{n}{2}\]

2.1.3. Fourier integral transforms

The usual application of the discrete Fourier transform is that of obtaining an approximation of the Fourier integral transform

\[
+\infty \\
F(s) = \int_{-\infty}^{+\infty} f(t) \exp(-i2\pi st) dt
\]

when \( f(t) \) is negligible outside some region \((0, c)\). Dividing the region into \( n \) equal intervals we have

\[
\begin{align*}
n-1 \\
c
\end{align*}
\]

\[
F(s)^{n-1} = \sum_{j=0}^{n-1} f \exp(-i2\pi sjc/n)
\]

and so

\[
\begin{align*}
n-1 \\
c
\end{align*}
\]

\[
F = \sum_{k=0}^{n-1} f \exp(-i2\pi jk/n)
\]

\[
k \\
n
\]
for \( k = 0, 1, \ldots, n-1 \), where \( f = f(jc/n) \) and \( F = F(k/c) \).

Hence the discrete Fourier transform gives an approximation to the Fourier integral transform in the region \( s = 0 \) to \( s = n/c \).

If the function \( f(t) \) is defined over some more general interval \((a, b)\), then the integral transform can still be approximated by the discrete transform provided a shift is applied to move the point \( a \) to the origin.

2.1.4. Convolutions and correlations

One of the most important applications of the discrete Fourier transform is to the computation of the discrete convolution or correlation of two vectors \( x \) and \( y \) defined (as in Brigham [1]) by:

\[
\text{convolution: } z = \sum_{j=0}^{n-1} x_j y_{j-k},
\]

\[
\text{correlation: } w = \sum_{j=0}^{n-1} x_j y_{j+k}.
\]

(Here \( x \) and \( y \) are assumed to be periodic with period \( n \).)

Under certain circumstances (see Brigham [1]) these can be used as approximations to the convolution or correlation integrals defined by:

\[
z(s) = \int_{-\infty}^{+\infty} x(t)y(s-t)dt
\]

\[
w(s) = \int_{-\infty}^{+\infty} x(t)y(s+t)dt, \quad -\infty < s < +\infty.
\]
For more general advice on the use of Fourier transforms, see Hamming [2]; more detailed information on the fast Fourier transform algorithm can be found in Van Loan [3] and Brigham [1].

2.2. References


3. Recommendations on Choice and Use of Routines

3.1. One-dimensional Fourier Transforms

The choice of routine is determined first of all by whether the data values constitute a real, Hermitian or general complex sequence. It is wasteful of time and storage to use an inappropriate routine.

Two groups, each of three routines, are provided

<table>
<thead>
<tr>
<th></th>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real sequences</td>
<td>C06EAF</td>
<td>C06FPF</td>
</tr>
<tr>
<td>Hermitian sequences</td>
<td>C06EBF</td>
<td>C06FQF</td>
</tr>
<tr>
<td>General complex sequences</td>
<td>C06ECF</td>
<td>C06FRF</td>
</tr>
</tbody>
</table>

Group 1 routines each compute a single transform of length n, without requiring any extra working storage. The Group 1 routines impose some restrictions on the value of n, namely that no prime factor of n may exceed 19 and the total number of prime factors (including repetitions) may not exceed 20 (though the latter restriction only becomes relevant when n>10).

Group 2 routines are designed to perform several transforms in a single call, all with the same value of n. They do however require more working storage. Even on scalar processors, they may be somewhat faster than repeated calls to Group 1 routines because of reduced overheads and because they pre-compute and
store the required values of trigonometric functions. Group 2 routines impose no practical restrictions on the value of \( n \); however the fast Fourier transform algorithm ceases to be ‘fast’ if applied to values of \( n \) which cannot be expressed as a product of small prime factors. All the above routines are particularly efficient if the only prime factors of \( n \) are 2, 3 or 5.

If extensive use is to be made of these routines, users who are concerned about efficiency are advised to conduct their own timing tests.

To compute inverse discrete Fourier transforms the above routines should be used in conjunction with the utility routines C06GBF, C06GCF and C06QPF which form the complex conjugate of a Hermitian or general sequence of complex data values.

### 3.2. Multi-dimensional Fourier Transforms

C06FUF computes a 2-dimensional discrete Fourier transform of a 2-dimensional sequence of complex data values. This is defined by

\[
    z = \frac{1}{n} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} z_{jk} e^{-i \frac{2\pi j k}{n}}.
\]

### 3.3. Convolution and Correlation

C06EKF computes either the discrete convolution or the discrete correlation of two real vectors.

### 3.4. Index

Complex conjugate, complex sequence C06GCF
Hermitian sequence C06GBF
multiple Hermitian sequences C06QPF
Complex sequence from Hermitian sequences C06GSF
Convolution or Correlation real vectors C06EKF
Discrete Fourier Transform two-dimensional complex sequence C06FUF
one-dimensional, multiple transforms complex sequence C06FRF
Hermitian sequence C06QPF
real sequence C06FPF
one-dimensional, single transforms
complex sequence C06ECF
Hermitian sequence C06EBF
real sequence C06EAF

C06 -- Summation of Series
Chapter C06

Contents -- C06

Summation of Series

C06EAF Single 1-D real discrete Fourier transform, no extra workspace
C06EBF Single 1-D Hermitian discrete Fourier transform, no extra workspace
C06ECF Single 1-D complex discrete Fourier transform, no extra workspace
C06Ekf Circular convolution or correlation of two real vectors, no extra workspace
C06FPF Multiple 1-D real discrete Fourier transforms
C06FQF Multiple 1-D Hermitian discrete Fourier transforms
C06FRF Multiple 1-D complex discrete Fourier transforms
C06FUF 2-D complex discrete Fourier transform
C06GBF Complex conjugate of Hermitian sequence
C06GCF Complex conjugate of complex sequence
C06GQF Complex conjugate of multiple Hermitian sequences
C06GSF Convert Hermitian sequences to general complex sequences

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is
不包括在数值算法基本库中。

1. 目的

C06EAF计算一个实数序列的离散傅里叶变换。 (无需额外工作空间。

2. 规格

```fortran
SUBROUTINE C06EAF (X, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N)
```

3. 描述

给定一个包含n个实数序列的序列x, 对于j=0,1,...,n-1，

这个程序计算其离散傅里叶变换为:

\[
    z_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp\left(-i \frac{2\pi jk}{n}\right), \quad k=0,1,...,n-1.
\]

（注意在定义中使用的1/√n的尺度因子。）通过调用C06GBF可以计算其逆离散傅里叶变换。

对于计算逆离散傅里叶变换，上述程序应先被调用，得到的结果：

\[
    w_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} \overline{x_j} \exp\left(+i \frac{2\pi jk}{n}\right), \quad k=0,1,...,n-1.
\]

（注：\(\overline{z_k}\)是序列的共轭。）
complex conjugates of the $z_k$.

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of $n$ (see Section 5).

4. References


5. Parameters

1: $X(N)$ -- DOUBLE PRECISION array Input/Output
On entry: if $X$ is declared with bounds (0:N-1) in the (sub)program from which C06EAF is called, then $X(j)$ must contain $x_j$, for $j=0,1,...,n-1$. On exit: the discrete Fourier transform stored in Hermitian form. If the components of the transform $z_k$ are written as $a_k + ib_k$ and if $X$ is declared with bounds (0:N-1) in the (sub)program from which C06EAF is called, then for $0\leq k\leq n/2$, $a_k$ is contained in $X(k)$, and for $1\leq k\leq (n-1)/2$, $b_k$ is contained in $X(n-k)$. (See also Section 2.1.2 of the Chapter Introduction, and the Example Program.)

2: $N$ -- INTEGER Input
On entry: the number of data values, $n$. The largest prime factor of $N$ must not exceed 19, and the total number of prime factors of $N$, counting repetitions, must not exceed 20. Constraint: $N > 1$.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of $N$ is greater than 19.
CHAPTER 15. CHAPTER N

IFAIL= 2
    N has more than 20 prime factors.

IFAIL= 3
    N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to n*logn, but also depends on the factorization of n. The routine is somewhat faster than average if the only prime factors of n are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has several unpaired prime factors, i.e., if the 'square-free' part of n has several factors. For such values of n, routine C06FAF(*) (which requires an additional n elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of real data values, and prints their discrete Fourier transform (as computed by C06EAF), after expanding it from Hermitian form into a full complex sequence.

It then performs an inverse transform using C06GBF and C06EBF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
not included in the Foundation Library.

1. Purpose

C06EBF calculates the discrete Fourier transform of a Hermitian sequence of $n$ complex data values. (No extra workspace required.)

2. Specification

```fortran
SUBROUTINE C06EBF (X, N, IFAIL)
    INTEGER N, IFAIL
    DOUBLE PRECISION X(N)
```

3. Description

Given a Hermitian sequence of $n$ complex data values $z$ (i.e., a sequence such that $z$ is real and $z^\ast$ is the complex conjugate of $z$, for $j=1,2,...,n-1$) this routine calculates their discrete Fourier transform defined by:

$$
x_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(-\frac{2\pi i jk}{n}\right), \quad k=0,1,...,n-1.
$$

(Note the scale factor of $\frac{1}{\sqrt{n}}$ in this definition.) The transformed values $x_k$ are purely real (see also the chapter introduction).

To compute the inverse discrete Fourier transform defined by:

$$
y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(+\frac{2\pi i jk}{n}\right),
$$

this routine should be preceded by a call of C06GBF to form the complex conjugates of the $z_j$. 

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of n (see Section 5).

4. References


5. Parameters

1: $X(N)$ -- DOUBLE PRECISION array.Input/Output
On entry: the sequence to be transformed stored in Hermitian form. If the data values $z$ are written as $x + iy$, and if $X$ is declared with bounds $(0:N-1)$ in the subroutine from which C06EBF is called, then for $0 \leq j \leq n/2$, $x$ is contained in $X(j)$, and for $1 \leq j \leq (n-1)/2$, $y$ is contained in $X(n-j)$. (See also Section 2.1.2 of the Chapter Introduction and the Example Program.) On exit: the components of the discrete Fourier transform $x$. If $X$ is declared with bounds $(0:N-1)$ in the (sub)program from which C06EBF is called, then $x$ is stored in $X(k)$, for $k=0,1,\ldots,n-1$.

2: $N$ -- INTEGER Input
On entry: the number of data values, $n$. The largest prime factor of $N$ must not exceed 19, and the total number of prime factors of $N$, counting repetitions, must not exceed 20. Constraint: $N > 1$.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of \( N \) is greater than 19.

IFAIL= 2
   \( N \) has more than 20 prime factors.

IFAIL= 3
   \( N \leq 1 \).

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( n \log n \), but also depends on the factorization of \( n \). The routine is somewhat faster than average if the only prime factors of \( n \) are 2, 3 or 5; and fastest of all if \( n \) is a power of 2.

On the other hand, the routine is particularly slow if \( n \) has several unpaired prime factors, i.e., if the 'square-free' part of \( n \) has several factors. For such values of \( n \), routine C06FBF(*) (which requires an additional \( n \) elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of real data values which is assumed to be a Hermitian sequence of complex data values stored in Hermitian form. The input sequence is expanded into a full complex sequence and printed alongside the original sequence. The discrete Fourier transform (as computed by C06EBF) is printed out.

The program then performs an inverse transform using C06EAF and C06GBF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06ECF calculates the discrete Fourier transform of a sequence of n complex data values. (No extra workspace required.)

2. Specification

SUBROUTINE C06ECF (X, Y, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), Y(N)

3. Description

Given a sequence of n complex data values \( z_j \), for \( j=0,1,\ldots,n-1 \),
this routine calculates their discrete Fourier transform defined by:

\[
Z_k = \sum_{j=0}^{n-1} z_j e^{-i \frac{2\pi j k}{n}}, \quad k=0,1,\ldots,n-1.
\]

(Note the scale factor of \( \frac{1}{\sqrt{n}} \) in this definition.)

To compute the inverse discrete Fourier transform defined by:

\[
W_k = \sum_{j=0}^{n-1} z_j e^{i \frac{2\pi j k}{n}}, \quad k=0,1,\ldots,n-1.
\]

this routine should be preceded and followed by calls of C06GCF to form the complex conjugates of the \( z_j \) and the \( z_k \).

The routine uses the fast Fourier transform (FFT) algorithm.
(Brigham [1]). There are some restrictions on the value of n (see Section 5).

4. References


5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: if X is declared with bounds (0:N-1) in the (sub)program from which C06ECF is called, then X(j) must contain \( x \), the real part of \( z \), for \( j=0,1,...,n-1 \). On exit: the 
    
    real parts \( a \) of the components of the discrete Fourier 
    
    transform. If X is declared with bounds (0:N-1) in the (sub)program from which C06ECF is called, then \( a \) is contained in 
    
    \( X(k) \), for \( k=0,1,...,n-1 \).

2: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: if Y is declared with bounds (0:N-1) in the (sub)program from which C06ECF is called, then Y(j) must contain \( y \), the imaginary part of \( z \), for \( j=0,1,...,n-1 \). On exit: 
    
    the imaginary parts \( b \) of the components of the discrete 
    
    Fourier transform. If Y is declared with bounds (0:N-1) in the (sub)program from which C06ECF is called, then \( b \) is 
    
    contained in \( Y(k) \), for \( k=0,1,...,n-1 \).

3: N -- INTEGER Input
On entry: the number of data values, \( n \). The largest prime factor of \( N \) must not exceed 19, and the total number of prime factors of \( N \), counting repetitions, must not exceed 20. Constraint: \( N > 1 \).

4: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
CHAPTER 15. CHAPTER N

Errors detected by the routine:

IFAIL = 1
   At least one of the prime factors of N is greater than 19.

IFAIL = 2
   N has more than 20 prime factors.

IFAIL = 3
   N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to n*logn, but also depends on the factorization of n. The routine is somewhat faster than average if the only prime factors of n are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has several unpaired prime factors, i.e., if the 'square-free' part of n has several factors. For such values of n, routine C06FCF(*) (which requires an additional n real elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of complex data values and prints their discrete Fourier transform.

It then performs an inverse transform using C06GCF and C06ECF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06EKF calculates the circular convolution or correlation of two real vectors of period n. No extra workspace is required.

2. Specification

```fortran
SUBROUTINE C06EKF (JOB, X, Y, N, IFAIL)
    INTEGER JOB, N, IFAIL
    DOUBLE PRECISION X(N), Y(N)
```

3. Description

This routine computes:

if JOB = 1, the discrete convolution of x and y, defined by:

\[
z_k = \sum_{j=0}^{n-1} x_j y_{k-j} \quad (k = 0, \ldots, n-1)
\]

if JOB = 2, the discrete correlation of x and y defined by:

\[
w_k = \sum_{j=0}^{n-1} x_j y_{k+j} \quad (k = 0, \ldots, n-1)
\]

Here x and y are real vectors, assumed to be periodic, with period n, i.e., \(x = x = \ldots\); z and w are then also periodic with period n.

Note: this usage of the terms 'convolution' and 'correlation' is taken from Brigham [1]. The term 'convolution' is sometimes used to denote both these computations.

If x, y, z and w are the discrete Fourier transforms of these sequences,
\[ i.e., x = \frac{\exp(-i \frac{2\pi j k}{n})}{\sqrt{n}} \]

\[ \text{then } z = \frac{\sum_{k} x^k y^k}{\sqrt{n}} \]

\[ \text{and } w = \frac{\sum_{k} x^k y^{k+j}}{\sqrt{n}} \]

\[(\text{the bar denoting complex conjugate}).\]

This routine calls the same auxiliary routines as C06EAF and C06EBF to compute discrete Fourier transforms, and there are some restrictions on the value of \( n \).

4. References


5. Parameters

1: JOB -- INTEGER

\[ \text{Input} \]

\[ \text{JOB} = 1 \text{ or } 2. \]

\[ \text{On entry: the computation to be performed:} \]

\[ n-1 \]

\[ \text{if } \text{JOB} = 1, z = \sum_{j=0}^{n-1} x^j y^{k-j} \] (convolution);

\[ \text{if } \text{JOB} = 2, w = \sum_{j=0}^{n-1} x^j y^{k+j} \] (correlation).

\[ \text{Constraint: JOB = 1 or 2.} \]

2: X(N) -- DOUBLE PRECISION array

\[ \text{Input/Output} \]

\[ \text{On entry: the elements of one period of the vector x. If X} \]

\[ \text{is declared with bounds (0:N-1) in the (sub)program from} \]

\[ \text{which C06EKF is called, then X(j) must contain x , for} \]

\[ j = 0, 1, \ldots, n-1. \]

\[ \text{On exit: the corresponding elements of the discrete convolution or correlation.} \]
3: Y(N) -- DOUBLE PRECISION array  
   Input/Output  
   On entry: the elements of one period of the vector y. If Y  
   is declared with bounds (0:N-1) in the (sub)program from  
   which C06EKF is called, then Y(j) must contain y , for  
   \[ j = 0,1,\ldots,n-1 \]. On exit: the discrete Fourier transform of  
   the convolution or correlation returned in the array X; the  
   transform is stored in Hermitian form, exactly as described  
   in the document C06EAF.

4: N -- INTEGER  
   Input  
   On entry: the number of values, n, in one period of the  
   vectors X and Y. The largest prime factor of N must not  
   exceed 19, and the total number of prime factors of N,  

5: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   At least one of the prime factors of N is greater than 19.

IFAIL = 2
   N has more than 20 prime factors.

IFAIL = 3
   N <= 1.

IFAIL = 4
   JOB /= 1 or 2.

7. Accuracy

The results should be accurate to within a small multiple of the  
machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to  
n*\log n, but also depends on the factorization of n. The routine
is faster than average if the only prime factors are 2, 3 or 5; and fastest of all if \( n \) is a power of 2.

The routine is particularly slow if \( n \) has several unpaired prime factors, i.e., if the 'square free' part of \( n \) has several factors. For such values of \( n \), routine C06FKF(*) is considerably faster (but requires an additional workspace of \( n \) elements).

9. Example

This program reads in the elements of one period of two real vectors \( x \) and \( y \) and prints their discrete convolution and correlation (as computed by C06EKF). In realistic computations the number of data values would be much larger.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
transforms of all the sequences defined by:

\[
\begin{align*}
\text{\( z = \sum_{j=0}^{\sqrt{n}} x * \exp(-i \frac{2\pi j k}{n}), \quad k=0,1,\ldots,n-1; \quad p=1,2,\ldots,m. \)}
\end{align*}
\]

(Note the scale factor \( \sqrt{n} \) in this definition.)

\[
\begin{align*}
\text{\( z = \sum_{j=0}^{\sqrt{n}} x * \exp(i \frac{2\pi j k}{n}), \quad k=0,1,\ldots,n-1; \quad p=1,2,\ldots,m. \)}
\end{align*}
\]

The transformed values \( z \) are complex, but for each value of \( p \), the \( z \) form a Hermitian sequence (i.e., \( z \) is the complex conjugate of \( z \)), so they are completely determined by \( mn \) real numbers (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term:

\[
\begin{align*}
\text{\( z = \sum_{j=0}^{\sqrt{n}} x * \exp(+i \frac{2\pi j k}{n}), \quad k=0,1,\ldots,n-1; \quad p=1,2,\ldots,m. \)}
\end{align*}
\]

To compute this form, this routine should be followed by a call to C06GQF to form the complex conjugates of the \( z \).

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special coding is provided for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as \( M \), the number of transforms to be computed in parallel, increases.

4. References

5. Parameters

1: M -- INTEGER  
   Input
   On entry: the number of sequences to be transformed, m.
   Constraint: M >= 1.

2: N -- INTEGER  
   Input
   On entry: the number of real values in each sequence, n.
   Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the data must be stored in X as if in a two-
   dimensional array of dimension (1:M,0:N-1); each of the m
   sequences is stored in a row of the array. In other words,
   if the data values of the pth sequence to be transformed are
   denoted by x , for j=0,1,...,n-1, then the mn elements of
   the array X must contain the values
       \[
       x_0, x_1, \ldots, x_{n-1}, \ldots, x_0, x_1, \ldots, x_{n-1},
       \]
   On exit: the m discrete Fourier transforms stored as if in
   a two-dimensional array of dimension (1:M,0:N-1). Each of
   the m transforms is stored in a row of the array in
   Hermitian form, overwriting the corresponding original
   sequence. If the n components of the discrete Fourier
   transform \( z^p \) are written as \( a^p + ib^p \), then for \( 0 \leq k < n/2 \), \( a_k^p \)
   is contained in \( X(p,k) \), and for \( 1 \leq k < (n-1)/2 \), \( b_k^p \)
   is contained in \( X(p,n-k) \). (See also Section 2.1.2 of the
   Chapter Introduction.)

4: INIT -- CHARACTER*1  
   Input
   On entry: if the trigonometric coefficients required to
   compute the transforms are to be calculated by the routine
   and stored in the array TRIG, then INIT must be set equal to
   'I' (Initial call).
   If INIT contains 'S' (Subsequent call), then the routine
   assumes that trigonometric coefficients for the specified
   value of n are supplied in the array TRIG, having been...
calculated in a previous call to one of C06FPF, C06FQF or C06FRF.

If INIT contains 'R' (Restart) then the routine assumes that trigonometric coefficients for the particular value of n are supplied in the array TRIG, but does not check that C06FPF, C06FQF or C06FRF have previously been called. This option allows the TRIG array to be stored in an external file, read in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is consistent with the array TRIG.

Constraint: INIT = 'I', 'S' or 'R'.

5: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

6: WORK(M*N) -- DOUBLE PRECISION array Workspace

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry M < 1.

IFAIL= 2
N < 1.

IFAIL= 3
INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
INIT = 'S', but none of C06FPF, C06FQF or C06FRF has previously been called.
CHAPTER 15. CHAPTER N

IFAIL= 5
    INIT = 'S' or 'R', but the array TRIG and the current value of N are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to nm*logn, but also depends on the factors of n. The routine is fastest if the only prime factors of n are 2, 3 and 5, and is particularly slow if n is a large prime, or has large prime factors.

9. Example

This program reads in sequences of real data values and prints their discrete Fourier transforms (as computed by C06FPF). The Fourier transforms are expanded into full complex form using C06GSF and printed. Inverse transforms are then calculated by calling C06GQF followed by C06FQF showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C06FQF(3NAG)  Foundation Library (12/10/92)  C06FQF(3NAG)

C06 -- Summation of Series
C06FQF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06FQF computes the discrete Fourier transforms of m Hermitian sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.
2. Specification

SUBROUTINE C06FQF (M, N, X, INIT, TRIG, WORK, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), TRIG(2*N), WORK(M*N)
CHARACTER*1 INIT

3. Description

Given m Hermitian sequences of n complex data values $z_j$, for $j=0,1,...,n-1$; $p=1,2,...,m$, this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

$$\frac{1}{n} \sum_{j=0}^{n-1} z_j \exp\left(-i \frac{2\pi pk}{n}\right), \quad k=0,1,...,n-1; \quad p=1,2,...,m.$$  
(Note the scale factor $\frac{1}{n}$ in this definition.)

The transformed values are purely real (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term

$$\frac{1}{n} \sum_{j=0}^{n-1} z_j \exp\left(i \frac{2\pi pk}{n}\right), \quad k=0,1,...,n-1; \quad p=1,2,...,m.$$  
To compute this form, this routine should be preceded by a call to C06GQF to form the complex conjugates of the $z_j$.

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special code is included for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and
it becomes especially fast as \( m \), the number of transforms to be computed in parallel, increases.

4. References


5. Parameters

1: \( M \) -- INTEGER Input
   On entry: the number of sequences to be transformed, \( m \).
   Constraint: \( M \geq 1 \).

2: \( N \) -- INTEGER Input
   On entry: the number of data values in each sequence, \( n \).
   Constraint: \( N \geq 1 \).

3: \( X(M,N) \) -- DOUBLE PRECISION array Input/Output
   On entry: the data must be stored in \( X \) as if in a two-dimensional array of dimension \((1:M,0:N-1)\); each of the \( m \) sequences is stored in a row of the array in Hermitian form.
   \[ x_{p,j} \quad \text{if } 0 \leq j \leq n/2, \quad y_{p,n-j} \quad \text{if } 1 \leq j \leq (n-1)/2, \]
   \( p \)
   If the \( n \) data values \( z \) are written as \( x+iy \), then for \( p \) \( j \) \( j \) \( p \)
   \( 0 \leq j \leq n/2, \ x \) is contained in \( X(p,j) \), and for \( 1 \leq j \leq (n-1)/2, \)
   \( j \)
   \( y \) is contained in \( X(p,n-j) \). (See also Section 2.1.2 of the Chapter Introduction.) On exit: the components of the \( m \) discrete Fourier transforms, stored as if in a two-dimensional array of dimension \((1:M,0:N-1)\). Each of the \( m \) transforms is stored as a row of the array, overwriting the corresponding original sequence. If the \( n \) components of the discrete Fourier transform are denoted by \( x_k \), for \( k = 0,1,\ldots,n-1 \), then the \( mn \) elements of the array \( X \) contain the values
   \[ x_0, x_1, \ldots, x_{n-1}, \quad x_0, x_1, \ldots, x_{n-1}, \quad x_0, x_1, \ldots, x_{n-1}, \quad \ldots, x_0, x_1, \ldots, x_{n-1} \]

4: \( INIT \) -- CHARACTER*1 Input
   On entry: if the trigonometric coefficients required to
compute the transforms are to be calculated by the routine and stored in the array TRIG, then INIT must be set equal to 'I' (Initial call).

If INIT contains 'S' (Subsequent call), then the routine assumes that trigonometric coefficients for the specified value of n are supplied in the array TRIG, having been calculated in a previous call to one of C06FPF, C06FQF or C06FRF.

If INIT contains 'R' (Restart), then the routine assumes that trigonometric coefficients for the particular value of N are supplied in the array TRIG, but does not check that C06FPF, C06FQF or C06FRF have previously been called. This option allows the TRIG array to be stored in an external file, read in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is compatible with the array TRIG. Constraint: INIT = 'I', 'S' or 'R'.

5: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
   On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

6: WORK(M*N) -- DOUBLE PRECISION array Workspace

7: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 1.

IFAIL= 2
   On entry N < 1.
CHAPTER 15.  CHAPTER N

IFAIL= 3
On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
On entry INIT = 'S', but none of C06FF, C06FQF and C06FRF has previously been called.

IFAIL= 5
On entry INIT = 'S' or 'R', but the array TRIG and the current value of n are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( nm \log n \), but also depends on the factors of n. The routine is fastest if the only prime factors of n are 2, 3 and 5, and is particularly slow if n is a large prime, or has large prime factors.

9. Example

This program reads in sequences of real data values which are assumed to be Hermitian sequences of complex data stored in Hermitian form. The sequences are expanded into full complex form using C06GSF and printed. The discrete Fourier transforms are then computed (using C06FQF) and printed out. Inverse transforms are then calculated by calling C06FF followed by C06GQF showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C06FRF(3NAG)  Foundation Library (12/10/92)  C06FRF(3NAG)
C06 -- Summation of Series  C06FRF
C06FRF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

C06FRF computes the discrete Fourier transforms of m sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.

2. Specification

```fortran
SUBROUTINE C06FRF (M, N, X, Y, INIT, TRIG, WORK, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), Y(M*N), TRIG(2*N), WORK(2*M*N)
CHARACTER*1 INIT
```

3. Description

Given m sequences of n complex data values \( z_j \), for \( j=0,1,...,n-1; \)

\( p=1,2,...,m \), this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

\[
\text{for } p=1,2,...,m: \quad z_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(-i \frac{2\pi jk}{n}\right), \quad k=0,1,...,n-1.
\]

(Note the scale factor \( \frac{1}{\sqrt{n}} \) in this definition.)

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term

\[
\text{for } p=1,2,...,m: \quad z_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(+i \frac{2\pi jk}{n}\right), \quad k=0,1,...,n-1.
\]

To compute this form, this routine should be preceded and followed by a call of C06GCF to form the complex conjugates of \( z_j^p \) and the \( z_k^p \).

The routine uses a variant of the fast Fourier transform (FFT)
algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special code is provided for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as m, the number of transforms to be computed in parallel, increases.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number of sequences to be transformed, m.
   Constraint: M >= 1.

2: N -- INTEGER Input
   On entry: the number of complex values in each sequence, n.
   Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real and imaginary parts of the complex data must be stored in X and Y respectively as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of each array. In other words, if the real parts of the pth sequence to be transformed are denoted by x , for j=0,1,...,n-1, then the mn elements of
   j
   the array X must contain the values
   
   1 2  m 1 2  m 1 2  m
   x ,x ,...,x , x ,x ,...,x , x ,x ,...,x .
   0 0 0 1 1  1 n-1 n-1 n-1

   On exit: X and Y are overwritten by the real and imaginary parts of the complex transforms.

4: Y(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: if the trigonometric coefficients required to compute the transforms are to be calculated by the routine and stored in the array TRIG, then INIT must be set equal to 'I' (Initial call).

5: INIT -- CHARACTER*1 Input
   On entry: if INIT contains 'S' (Subsequent call), then the routine
assumes that trigonometric coefficients for the specified
value of n are supplied in the array TRIG, having been
calculated in a previous call to one of C06FPF, C06FQF or
C06FRF.

If INIT contains 'R' (Restart) then the routine assumes that
trigonometric coefficients for the particular value of n are
supplied in the array TRIG, but does not check that C06FPF,
C06FQF or C06FRF have previously been called. This option
allows the TRIG array to be stored in an external file, read
in and re-used without the need for a call with INIT equal
to 'I'. The routine carries out a simple test to check that
the current value of n is compatible with the array TRIG.
Constraint: INIT = 'I', 'S' or 'R'.

6: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
On entry: if INIT = 'S' or 'R', TRIG must contain the
required coefficients calculated in a previous call of the
routine. Otherwise TRIG need not be set. On exit: TRIG
contains the required coefficients (computed by the routine
if INIT = 'I').

7: WORK(2*M*N) -- DOUBLE PRECISION array Workspace

8: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry M < 1.

IFAIL= 2
On entry N < 1.

IFAIL= 3
On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
On entry INIT = 'S', but none of C06FPF, C06FQF and C06FRF
has previously been called.

IFAIL = 5
On entry INIT = 'S' or 'R', but the array TRIG and the current value of n are inconsistent.

7. Accuracy
Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments
The time taken by the routine is approximately proportional to nm*logn, but also depends on the factors of n. The routine is fastest if the only prime factors of n are 2, 3 and 5, and is particularly slow if n is a large prime, or has large prime factors.

9. Example
This program reads in sequences of complex data values and prints their discrete Fourier transforms (as computed by C06FRF). Inverse transforms are then calculated using C06GCF and C06FRF and printed out, showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C06FUF(3NAG) Foundation Library (12/10/92) C06FUF(3NAG)

C06 -- Summation of Series
C06FUF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
C06FUF computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values. This routine is designed to be particularly efficient on vector processors.
2. Specification

SUBROUTINE C06FUF (M, N, X, Y, INIT, TRIGM, TRIGN, WORK, IFAIL)

INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), Y(M*N), TRIGM(2*M), TRIGN(2*N), WORK(2*M*N)
CHARACTER*1 INIT

3. Description

This routine computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values $z_{jk}$, where $j=0,1,...,m-1$, $k=0,1,...,n-1$.

The discrete Fourier transform is here defined by:

$$z_{jk} = \frac{1}{\sqrt{mn}} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} z_{jk} \exp(-2\pi i \left( \frac{j}{m} + \frac{k}{n} \right)),$$

where $k=0,1,...,m-1$, $j=0,1,...,n-1$.

(Note the scale factor of $\sqrt{mn}$ in this definition.)

To compute the inverse discrete Fourier transform, defined with $\exp(+2\pi i \left( \frac{j}{m} + \frac{k}{n} \right))$ in the above formula instead of $\exp(-2\pi i \left( \frac{j}{m} + \frac{k}{n} \right))$, this routine should be preceded and followed by calls of C06GCF to form the complex conjugates of the data values and the transform.

This routine calls C06FRF to perform multiple one-dimensional discrete Fourier transforms by the fast Fourier transform (FFT) algorithm in Brigham [1]. It is designed to be particularly efficient on vector processors.

4. References

5. Parameters

1: M -- INTEGER  
   Input
   On entry: the number of rows, m, of the arrays X and Y.
   Constraint: M >= 1.

2: N -- INTEGER  
   Input
   On entry: the number of columns, n, of the arrays X and Y.
   Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the real and imaginary parts of the complex data
   values must be stored in arrays X and Y respectively. If X
   and Y are regarded as two-dimensional arrays of dimension
   (0:M-1,0:N-1), then X(j ,j ) and Y(j ,j ) must contain the
   1 2 1 2
   real and imaginary parts of z . On exit: the real and
   j j
   1 2
   imaginary parts respectively of the corresponding elements
   of the computed transform.

4: Y(M,N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the real and imaginary parts of the complex data
   values must be stored in arrays X and Y respectively. If X
   and Y are regarded as two-dimensional arrays of dimension
   (0:M-1,0:N-1), then X(j ,j ) and Y(j ,j ) must contain the
   1 2 1 2
   real and imaginary parts of z . On exit: the real and
   j j
   1 2
   imaginary parts respectively of the corresponding elements
   of the computed transform.

5: INIT -- CHARACTER*1  
   Input
   On entry: if the trigonometric coefficients required to
   compute the transforms are to be calculated by the routine
   and stored in the arrays TRIGM and TRIGN, then INIT must be
   set equal to 'I', (Initial call).

   If INIT contains 'S', (Subsequent call), then the routine
   assumes that trigonometric coefficients for the specified
   values of m and n are supplied in the arrays TRIGM and
   TRIGN, having been calculated in a previous call to the
   routine.

   If INIT contains 'R', (Restart), then the routine assumes
   that trigonometric coefficients for the particular values of
   m and n are supplied in the arrays TRIGM and TRIGN, but does
   not check that the routine has previously been called. This
   option allows the TRIGM and TRIGN arrays to be stored in an
   external file, read in and re-used without the need for a
   call with INIT equal to 'I'. The routine carries out a
   simple test to check that the current values of m and n are
   compatible with the arrays TRIGM and TRIGN. Constraint: INIT
6: \text{TRIGM}(2\times M) -- DOUBLE PRECISION array \hspace{1cm} \text{Input/Output}

7: \text{TRIGN}(2\times N) -- DOUBLE PRECISION array \hspace{1cm} \text{Input/Output}

On entry: if INIT = 'S' or 'R', TRIGM and TRIGN must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIGM and TRIGN need not be set.

If \( m=n \) the same array may be supplied for TRIGM and TRIGN.

On exit: TRIGM and TRIGN contain the required coefficients (computed by the routine if INIT = 'I').

8: \text{WORK}(2\times M\times N) -- DOUBLE PRECISION array \hspace{1cm} \text{Workspace}

9: IFAIL -- INTEGER \hspace{1cm} \text{Input/Output}

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry \( M < 1 \).

IFAIL= 2
On entry \( N < 1 \).

IFAIL= 3
On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
On entry INIT = 'S', but C06FUF has not previously been called.

IFAIL= 5
On entry INIT = 'S' or 'R', but at least one of the arrays TRIGM and TRIGN is inconsistent with the current value of M or N.

7. Accuracy
Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( mn \times \log(mn) \), but also depends on the factorization of the individual dimensions \( m \) and \( n \). The routine is somewhat faster than average if their only prime factors are 2, 3 or 5; and fastest of all if they are powers of 2.

9. Example

This program reads in a bivariate sequence of complex data values and prints the two-dimensional Fourier transform. It then performs an inverse transform and prints the sequence so obtained, which may be compared to the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

SUBROUTINE C06GBF (X, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N)

This is a utility routine for use in conjunction with C06EAF, C06EBF, C06FAF(*) or C06FBF(*) to calculate inverse discrete
Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
   On entry: if the data values z are written as x +iy and
   j j j
   if X is declared with bounds (0:N-1) in the (sub)program
   from which C06GBF is called, then for 0<=j<=n/2, X(j) must
   contain x (=x ), while for n/2<j<=n-1, X(j) must contain
   j n-j
   -y (=y ). In other words, X must contain the Hermitian
   j n-j
   sequence in Hermitian form. (See also Section 2.1.2 of the
   Chapter Introduction). On exit: the imaginary parts y are
   j
   negated. The real parts x are not referenced.

2: N -- INTEGER Input
   On entry: the number of data values, n. Constraint: N >= 1.

3: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example
This program reads in a sequence of real data values, calls C06EAF followed by C06GBF to compute their inverse discrete Fourier transform, and prints this after expanding it from Hermitian form into a full complex sequence.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
C06GCF forms the complex conjugate of a sequence of n data values.

2. Specification
SUBROUTINE C06GCF (Y, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION Y(N)

3. Description
This is a utility routine for use in conjunction with C06ECF or C06FCF(*) to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References
None.

5. Parameters
1: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: if Y is declared with bounds (0:N-1) in the (sub) program which C06GCF is called, then Y(j) must contain the imaginary part of the jth data value, for 0<=j<=n-1. On exit: these values are negated.
2: N -- INTEGER  
   Input  
   On entry: the number of data values, n. Constraint: N >= 1.

3: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example

This program reads in a sequence of complex data values and  
prints their inverse discrete Fourier transform as computed by  
calling C06GCF, followed by C06ECF and C06GCF again.

The example program is not reproduced here. The source code for  
all example programs is distributed with the NAG Foundation  
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C06GQF(3NAG)  Foundation Library (12/10/92)  C06GQF(3NAG)

C06 -- Summation of Series  
   C06GQF  
   C06GQF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for  
your implementation to check implementation-dependent details.  
The symbol (*) after a NAG routine name denotes a routine that is  
not included in the Foundation Library.

1. Purpose
C06GQF forms the complex conjugates of m Hermitian sequences, each containing n data values.

2. Specification

SUBROUTINE C06GQF (M, N, X, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N)

3. Description

This is a utility routine for use in conjunction with C06FPF and C06FQF to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: M -- INTEGER Input
   On entry: the number of Hermitian sequences to be conjugated, m. Constraint: M >= 1.

2: N -- INTEGER Input
   On entry: the number of data values in each Hermitian sequence, n. Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: the data must be stored in array X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array in Hermitian form. If the n data values z are written as x +iy, then
     \[ x = X(p,j) \quad \text{for} \quad 0 \leq j \leq n/2, \quad y = X(p,n-j) \quad \text{for} \quad 1 \leq j \leq (n-1)/2, \]
   (See also Section 2.1.2 of the Chapter Introduction.) On exit: the imaginary parts \( y \) are negated. The real parts \( x \) are not referenced.

4: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 1.

IFAIL= 2
   On entry N < 1.

7. Accuracy

Exact.

8. Further Comments

None.

9. Example

This program reads in sequences of real data values which are
assumed to be Hermitian sequences of complex data stored in
Hermitian form. The sequences are expanded into full complex form
using C06GSF and printed. The sequences are then conjugated
(using C06GQF) and the conjugated sequences are expanded into
complex form using C06GSF and printed out.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C06GSF(3NAG) Foundation Library (12/10/92) C06GSF(3NAG)
C06 -- Summation of Series C06GSF
C06GSF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C06GSF takes \( m \) Hermitian sequences, each containing \( n \) data values, and forms the real and imaginary parts of the \( m \) corresponding complex sequences.

2. Specification

```fortran
SUBROUTINE C06GSF (M, N, X, U, V, IFAIL)
  INTEGER M, N, IFAIL
  DOUBLE PRECISION X(M*N), U(M*N), V(M*N)
```

3. Description

This is a utility routine for use in conjunction with C06FPF and C06FQF (see the Chapter Introduction).

4. References

None.

5. Parameters

1: \( M \) -- INTEGER Input
   On entry: the number of Hermitian sequences, \( m \), to be converted into complex form. Constraint: \( M \geq 1 \).

2: \( N \) -- INTEGER Input
   On entry: the number of data values, \( n \), in each sequence. Constraint: \( N \geq 1 \).

3: \( X(M,N) \) -- DOUBLE PRECISION array Input
   On entry: the data must be stored in \( X \) as if in a two-dimensional array of dimension \( (1:M,0:N-1) \); each of the \( m \) sequences is stored in a row of the array in Hermitian form.
   
   If the \( n \) data values \( z \) are written as \( x + iy \), then for \( 0 \leq j \leq \lfloor n/2 \rfloor \), \( x \) is contained in \( X(p,j) \), and for \( 1 \leq j \leq (n-1)/2 \), \( y \) is contained in \( X(p,n-j) \). (See also Section 2.1.2 of the Chapter Introduction.)

4: \( U(M,N) \) -- DOUBLE PRECISION array Output
5: V(M,N) -- DOUBLE PRECISION array Output
   On exit: the real and imaginary parts of the m sequences of
   length n, are stored in U and V respectively, as if in two-
   dimensional arrays of dimension (1:M,0:N-1); each of the m
   sequences is stored as if in a row of each array. In other
   words, if the real parts of the pth sequence are denoted by
   \( x_p \), for \( j=0,1,\ldots,n-1 \) then the mn elements of the array U
   \( j \)
   contain the values
   \[
   x_0, x_1, \ldots, x_n, x_1, x_2, \ldots, x_n, \ldots, x_{m-1}, x_m, \ldots, x_{m+n-1}, x_{m+n-1}, \ldots, x_{mn-1}, x_{mn}, \ldots, x_{mn+n-1}, \ldots
   \]

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 1.

IFAIL= 2
   On entry N < 1.

7. Accuracy

Exact.

8. Further Comments

None.

9. Example

This program reads in sequences of real data values which are
assumed to be Hermitian sequences of complex data stored in
Hermitian form. The sequences are then expanded into full complex
form using C06GSF and printed.
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

NagSeriesSummationPackage (NAGC06)

Exports:
c06eaf c06ebf c06ecf c06ekf c06fpf
c06fqf c06frf c06fuf c06gbf c06gcf
c06gsf

— package NAGC06 NagSeriesSummationPackage —

)abbrev package NAGC06 NagSeriesSummationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:30 1994
++ Description:
++ This package uses the NAG Library to calculate the discrete Fourier
++ transform of a sequence of real or complex data values, and
++ applies it to calculate convolutions and correlations.

NagSeriesSummationPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
c06eaf : (Integer,Matrix DoubleFloat, Integer) -> Result
++ c06eaf(n,x,ifail)
++ calculates the discrete Fourier transform of a sequence of
++ n real data values. (No extra workspace required.)
++ See \downlink{Manual Page}{manpageXXc06eaf}. 
c06ebf : (Integer, Matrix DoubleFloat, Integer) -> Result
  ** c06ebf(n,x,ifail)
  ** calculates the discrete Fourier transform of a Hermitian
  ** sequence of n complex data values. (No extra workspace required.)
  ** See \downlink{Manual Page}{manpageXXc06ebf}.
c06ecf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
  ** c06ecf(n,x,y,ifail)
  ** calculates the discrete Fourier transform of a sequence of
  ** n complex data values. (No extra workspace required.)
  ** See \downlink{Manual Page}{manpageXXc06ecf}.
c06ekf : (Integer, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
  ** c06ekf(job,n,x,y,ifail)
  ** calculates the circular convolution of two
  ** real vectors of period n. No extra workspace is required.
  ** See \downlink{Manual Page}{manpageXXc06ekf}.
c06fpf : (Integer, Integer, String, Matrix DoubleFloat, Integer) -> Result
  ** c06fpf(m,n,init,x,trig,ifail)
  ** computes the discrete Fourier transforms of m sequences,
  ** each containing n real data values. This routine is designed to
  ** be particularly efficient on vector processors.
  ** See \downlink{Manual Page}{manpageXXc06fpf}.
c06fqf : (Integer, Integer, String, Matrix DoubleFloat, Integer) -> Result
  ** c06fqf(m,n,init,x,trig,ifail)
  ** computes the discrete Fourier transforms of m Hermitian
  ** sequences, each containing n complex data values. This routine is
  ** designed to be particularly efficient on vector processors.
  ** See \downlink{Manual Page}{manpageXXc06fqf}.
c06frf : (Integer, Integer, String, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
  ** c06frf(m,n,init,x,y,trig,ifail)
  ** computes the discrete Fourier transforms of m sequences,
  ** each containing n complex data values. This routine is designed
  ** to be particularly efficient on vector processors.
  ** See \downlink{Manual Page}{manpageXXc06frf}.
c06fuf : (Integer, Integer, String, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
  ** c06fuf(m,n,init,x,y,trigm,trign,ifail)
  ** computes the two-dimensional discrete Fourier transform of
  ** a bivariate sequence of complex data values. This routine is
  ** designed to be particularly efficient on vector processors.
  ** See \downlink{Manual Page}{manpageXXc06fuf}.
c06gbf : (Integer, Matrix DoubleFloat, Integer) -> Result
  ** c06gbf(n,x,ifail)
  ** forms the complex conjugate of n
  ** data values.
  ** See \downlink{Manual Page}{manpageXXc06gbf}.
c06gcf : (Integer, Matrix DoubleFloat, Integer) -> Result
  ++ c06gcf(n, y, ifail)
  ++ forms the complex conjugate of a sequence of n data
  ++ values.
  ++ See `\downlink{Manual Page}{manpageXXc06gcf}`.

c06gqf : (Integer, Integer, Matrix DoubleFloat, Integer) -> Result
  ++ c06gqf(m, n, x, ifail)
  ++ forms the complex conjugates,
  ++ each containing n data values.
  ++ See `\downlink{Manual Page}{manpageXXc06gqf}`.

c06gsf : (Integer, Integer, Matrix DoubleFloat, Integer) -> Result
  ++ c06gsf(m, n, x, ifail)
  ++ takes m Hermitian sequences, each containing n data
  ++ values, and forms the real and imaginary parts of the m
  ++ corresponding complex sequences.
  ++ See `\downlink{Manual Page}{manpageXXc06gsf}`.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(Matrix DoubleFloat)

c06eaf(nArg: Integer, xArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "c06eaf",_
    ["n"::S,"ifail"::S,"x"::S]$Lisp,_
    []$Lisp,_
    [["double"::S, ["x"::S,"n"::S]$Lisp]$Lisp,$Lisp_,
      ["integer"::S,"n"::S,"ifail"::S]$Lisp_]
    ]$Lisp,_
    ["x"::S,"ifail"::S]$Lisp,_
    [[[nArg::Any, ifailArg::Any, xArg::Any ]]_
      @List Any]$Lisp]$Lisp)_
  pretend List (Record(key:Symbol,entry:Any))]$Result


c06ebf(nArg: Integer, xArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "c06ebf",_
    ["n"::S,"ifail"::S,"x"::S]$Lisp,_
    []$Lisp,_
    [["double"::S, ["x"::S,"n"::S]$Lisp]$Lisp,$Lisp_]
    )$Lisp,$Lisp)
  pretend List (Record(key:Symbol,entry:Any))]$Result

c06ecf(nArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06ecf",_[
"n"::S,"ifail"::S,"x"::S,"y"::S]$Lisp,_ []$Lisp,_
["double"::S,["x"::S,"n"::S]$Lisp,["y"::S,"n"::S]$Lisp_]
$Lisp,$Lisp_,
["integer"::S,"n"::S,"ifail"::S]$Lisp_]
$Lisp_,$Lisp_,
["x"::S,"y"::S,"ifail"::S]$Lisp_]
[(nArg::Any,ifailArg::Any,xArg::Any,yArg::Any )_ @List Any$Lisp$Lisp_]
pretend List (Record(key:Symbol,entry:Any)))$Result


c06ekf(jobArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
yArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06ekf",_[
["double"::S,["x"::S,"n"::S]$Lisp,["y"::S,"n"::S]$Lisp_]
$Lisp_,$Lisp_,
$Lisp_,$Lisp_,
["x"::S,"y"::S,"ifail"::S]$Lisp_]
[(jobArg::Any,nArg::Any,ifailArg::Any,xArg::Any,yArg::Any )_ @List Any$Lisp$Lisp_]
pretend List (Record(key:Symbol,entry:Any)))$Result


c06fpf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,trigArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06fpf",_[
["work"::S]$Lisp_,$Lisp_,
["double"::S,["x"::S,"*"::S,"m"::S,"n"::S]$Lisp]$Lisp_,
["*"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_]
[Lisp,Lisp_,
["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_]
["character"::S,"init"::S]$Lisp_]
$Lisp_,

CHAPTER 15.  CHAPTER N

Lisp
c06fqf(mArg: Integer, nArg: Integer, initArg: String,
  xArg: Matrix DoubleFloat, trigArg: Matrix DoubleFloat,
  ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp,
  "c06fqf",
  ["work"::S]$Lisp,
  ["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["trig"::S,["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,
  ["work"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,$Lisp,
  ["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp,
  ["character"::S,"init"::S]$Lisp
  ]$Lisp,
  ["x"::S,"trig"::S,"ifail"::S]$Lisp,
  [(mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
    xArg::Any,trigArg::Any )]
  @List Any]$Lisp)$Lisp) pretend List (Record(key:Symbol,entry:Any))$Result

c06frf(mArg: Integer, nArg: Integer, initArg: String,
  xArg: Matrix DoubleFloat, yArg: Matrix DoubleFloat,
  trigArg: Matrix DoubleFloat,
  ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp,
  "c06frf",
  "work"::S]$Lisp,
  ["work"::S]$Lisp,
  ["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["y"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,
  ["work"::S,["*"::S,"*"::S,"m"::S,"n"::S]$Lisp]$Lisp,$Lisp,
  ["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp,
  ["character"::S,"init"::S]$Lisp,$Lisp,
  ["x"::S,"y"::S,"trig"::S,"ifail"::S]$Lisp,
  [(mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
    xArg::Any,yArg::Any,trigArg::Any )]
  @List Any]$Lisp)$Lisp) pretend List (Record(key:Symbol,entry:Any))$Result

c06fuf(mArg: Integer, nArg: Integer, initArg: String,
  xArg: Matrix DoubleFloat, yArg: Matrix DoubleFloat,
  trigArg: Matrix DoubleFloat,
  trignArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
c06fuf(nArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,yArg::Any,trigmArg::Any,trignArg::Any): Result ==
[(invokeNagman(NIL$Lisp,_,
"c06fuf",_,
"trign":S,"work":S]$Lisp,_,
["work":S]$Lisp,_,
["double":S,["x":S,"m":S,"n":S]$Lisp]$Lisp,
["y":S,["m":S,"n":S]$Lisp]$Lisp,["trigm":S,_
["trign":S]$Lisp,$Lisp,
[*::S,2$Lisp,"m":S]$Lisp]$Lisp,["work":S,_
["m":S,"n":S]$Lisp]$Lisp,$Lisp,
,"character":S,"init":S]$Lisp,_,
([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
yArg::Any,trigmArg::Any,trignArg::Any ])
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c06gbf(nArg::Integer,xArg::Matrix DoubleFloat,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"c06gbf",_,
[n":S,"ifail":S,"x":S]$Lisp,_,
[])$Lisp,_,
["double":S,["x":S,"n":S]$Lisp]$Lisp,
["integer":S,"n":S,"ifail":S]$Lisp,$Lisp,
["x":S,"ifail":S]$Lisp,$Lisp,
([nArg::Any,ifailArg::Any,xArg::Any ])
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c06gcf(nArg::Integer,yArg::Matrix DoubleFloat,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"c06gcf",_,
[n":S,"ifail":S,"y":S]$Lisp,_,
[])$Lisp,_,
["double":S,["y":S,"n":S]$Lisp]$Lisp,
["integer":S,"n":S,"ifail":S]$Lisp,$Lisp,
[y":S,"ifail":S]$Lisp,$Lisp,
([nArg::Any,ifailArg::Any,yArg::Any ])
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c06gqf(mArg::Integer,nArg::Integer,xArg::Matrix DoubleFloat,_,
ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"c06gqf",_,
"trign":S,"work":S]$Lisp,_,
["work":S]$Lisp,_,
["double":S,["x":S,"m":S,"n":S]$Lisp]$Lisp,
["y":S,["m":S,"n":S]$Lisp,$Lisp,["trigm":S,_
["trign":S]$Lisp,$Lisp,$Lisp,
[*::S,2$Lisp,"m":S]$Lisp]$Lisp,["work":S,_
["m":S,"n":S]$Lisp]$Lisp]$Lisp,
,"character":S,"init":S]$Lisp,_,
([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
yArg::Any,trigmArg::Any,trignArg::Any ])
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c06gsf(mArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06gsf",_
"u":S,"v":S]$Lisp,_
"double":S,["x":S,["*:S,"m":S,"n":S]$Lisp]$Lisp,_
"u":S,["*:S,"m":S,"n":S]$Lisp]$Lisp,_
"v":S,["*:S,"m":S,"n":S]$Lisp]$Lisp,$Lisp,_
"integer":S,"m":S,"n":S,"ifail":S]$Lisp_]
$Lisp_,
"u":S,"v":S,"ifail":S]$Lisp,
[(mArg::Any,nArg::Any,ifailArg::Any,xArg::Any ])_
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

package NAGS NagSpecialFunctionsPackage

— NagSpecialFunctionsPackage.input —

)set break resume
)sys rm -f NagSpecialFunctionsPackage.output
)spool NagSpecialFunctionsPackage.output
This package uses the NAG Library to compute some commonly occurring physical and mathematical functions.

1. Scope of the Chapter

This chapter is concerned with the provision of some commonly occurring physical and mathematical functions.

2. Background to the Problems

The majority of the routines in this chapter approximate real-valued functions of a single real argument, and the techniques involved are described in Section 2.1. In addition the chapter contains routines for elliptic integrals (see Section 2.2), Bessel and Airy functions of a complex argument (see Section 2.3), and exponential of a complex argument.

2.1. Functions of a Single Real Argument

Most of the routines for functions of a single real argument have been based on truncated Chebyshev expansions. This method of approximation was adopted as a compromise between the conflicting requirements of efficiency and ease of implementation on many different machine ranges. For details of the reasons behind this choice and the production and testing procedures followed in constructing this chapter see Schonfelder [7].

Basically if the function to be approximated is \( f(x) \), then for \( x \) in \([a, b]\) an approximation of the form
\[ f(x) = g(x) \sum_{r=0}^{n} C_r T_r(t) \]

is used, (\( \sum \) denotes, according to the usual convention, a summation in which the first term is halved), where \( g(x) \) is some suitable auxiliary function which extracts any singularities, asymptotes and, if possible, zeros of the function in the range in question and \( t = t(x) \) is a mapping of the general range \([a, b]\) to the specific range \([-1, +1]\) required by the Chebyshev polynomials, \( T_r(t) \). For a detailed description of the properties of the Chebyshev polynomials see Clenshaw [5] and Fox and Parker [6].

The essential property of these polynomials for the purposes of function approximation is that \( T_r(t) \) oscillates between \([-1, +1]\) and it takes its extreme values \( n+1 \) times in the interval \([-1, +1]\). Therefore, provided the coefficients \( C_r \) decrease in magnitude sufficiently rapidly the error made by truncating the Chebyshev expansion after \( n \) terms is approximately given by

\[ E(t) \approx C \sum_{r=0}^{n} T_r(t) \]

That is the error oscillates between \( \pm C \) and takes its extreme value \( n+1 \) times in the interval in question. Now this is just the condition that the approximation be a mini-max representation, one which minimizes the maximum error. By suitable choice of the interval, \([a, b]\), the auxiliary function, \( g(x) \), and the mapping of the independent variable, \( t(x) \), it is almost always possible to obtain a Chebyshev expansion with rapid convergence and hence truncations that provide near mini-max polynomial approximations to the required function. The difference between the true mini-max polynomial and the truncated Chebyshev expansion is seldom sufficiently great to be of significance.

The evaluation of the Chebyshev expansions follows one of two methods. The first and most efficient, and hence most commonly used, works with the equivalent simple polynomial. The second method, which is used on the few occasions when the first method proves to be unstable, is based directly on the truncated Chebyshev series and uses backward recursion to evaluate the sum.
For the first method, a suitably truncated Chebyshev expansion (truncation is chosen so that the error is less than the machine precision) is converted to the equivalent simple polynomial. That is we evaluate the set of coefficients $b$ such that

$$
y(t) = \sum_{r=0}^{n-1} b_r t^r = C T_n(t).
$$

The polynomial can then be evaluated by the efficient Horner’s method of nested multiplications,

$$
y(t) = (b_n + t(b_{n-1} + t(b_{n-2} + \ldots (b_0 + t \ldots )))) \ldots ).
$$

This method of evaluation results in efficient routines but for some expansions there is considerable loss of accuracy due to cancellation effects. In these cases the second method is used. It is well known that if

$$
b_{n-1} = C
$$

$$
b_{n-2} = 2tb_{n-1} + C
$$

$$
b_j = 2tb_j - b_{j+1} + C, \quad j=n-3,n-4,\ldots,0
$$

then

$$
y(t) = -(b_0 - b_1) t^{n-2} + 2b_2 t^{n-3} + \ldots - b_n t^0
$$

and this is always stable. This method is most efficiently implemented by using three variables cyclically and explicitly constructing the recursion.

That is,

$$
\begin{align*}
(\alpha) &= C \\
n-1 \\
(\beta) &= 2t(\alpha) + C \\
n-2 \\
(\gamma) &= 2t(\beta) - (\alpha) + C
\end{align*}
$$
\[
\begin{align*}
\alpha &= 2t\gamma - \beta + C \\
\beta &= 2t\alpha - \gamma + C \\
&\vdots
\end{align*}
\]

The auxiliary functions used are normally functions compounded of simple polynomial (usually linear) factors extracting zeros, and the primary compiler-provided functions, \(\sin, \cos, \ln, \exp, \sqrt{\cdot}\), which extract singularities and/or asymptotes or in some cases basic oscillatory behaviour, leaving a smooth well-behaved function to be approximated by the Chebyshev expansion which can therefore be rapidly convergent.

The mappings of \([a,b]\) to \([-1,+1]\) used, range from simple linear mappings to the case when \(b\) is infinite and considerable improvement in convergence can be obtained by use of a bilinear form of mapping. Another common form of mapping is used when the function is even, that is it involves only even powers in its expansion. In this case an approximation over the whole interval \([-a,a]\) can be provided using a mapping \(t=2(-a) - 1\). This embodies the evenness property but the expansion in \(t\) involves all powers and hence removes the necessity of working with an expansion with half its coefficients zero.

For many of the routines an analysis of the error in principle is given, viz, if \(E\) and \(\nabla\) are the absolute errors in function and argument and \((\epsilon)\) and \((\delta)\) are the corresponding relative errors, then

\[
E^\prime = |f'(x)|(\nabla)
\]

\[
E^\prime = |xf'(x)|(\delta)
\]

\[
|xf'(x)|
\]

\[
(\epsilon) = |(\delta) - f(x)|
\]

If we ignore errors that arise in the argument of the function by
propagation of data errors etc and consider only those errors that result from the fact that a real number is being represented in the computer in floating-point form with finite precision, then (\( \delta \)) is bounded and this bound is independent of the magnitude of \( x \); e.g. on an 11-digit machine

\[
-11 \\
|\( \delta \)|\( \leq \) 10 .
\]

(This of course implies that the absolute error \( \nabla = x(\delta) \) is also bounded but the bound is now dependent on \( x \)). However because of this the last two relations above are probably of more interest. If possible the relative error propagation is discussed; that is the behaviour of the error amplification factor \(|xf'(x)/f(x)|\) is described, but in some cases, such as near zeros of the function which cannot be extracted explicitly, absolute error in the result is the quantity of significance and here the factor \(|xf'(x)|\) is described. In general, testing of the functions has shown that their error behaviour follows fairly well these theoretical error behaviours. In regions, where the error amplification factors are less than or of the order of one, the errors are slightly larger than the above predictions. The errors are here limited largely by the finite precision of arithmetic in the machine but \( \epsilon \) is normally no more than a few times greater than the bound on \( \delta \). In regions where the amplification factors are large, order of ten or greater, the theoretical analysis gives a good measure of the accuracy obtainable.

It should be noted that the definitions and notations used for the functions in this chapter are all taken from Abramowitz and Stegun [1]. Users are strongly recommended to consult this book for details before using the routines in this chapter.

2.2. Approximations to Elliptic Integrals

The functions provided here are symmetrised variants of the classic elliptic integrals. These alternative definitions have been suggested by Carlson (see [2], [3] and [4]) and he also developed the basic algorithms used in this chapter.

The standard integral of the first kind is represented by

\[
\int_{\text{infty}}^{1} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]

\[
R(x,y,z) = - \int_{F}^{2} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]

\[
F = \int_{0}^{1} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]
where \( x, y, z \geq 0 \) and at most one may be equal to zero.

The normalisation factor, \(-\), is chosen so as to make

\[
R(x,x,x) = 1/\sqrt{x}.
\]

If any two of the variables are equal, \( R \) degenerates into the second function

\[
\frac{1}{
\int_0^{\infty} \frac{dt}{\sqrt{t+x(t+y)}}
}
\]

where the argument restrictions are now \( x \geq 0 \) and \( y \neq 0 \).

This function is related to the logarithm or inverse hyperbolic functions if \( 0 < y < x \), and to the inverse circular functions if \( 0 \leq x \leq y \).

The integrals of the second kind are defined by

\[
\frac{1}{
\int_0^{\infty} \frac{dt}{\sqrt{t+x(t+y)(t+z)}}
}
\]

with \( z > 0 \), \( x \geq 0 \) and \( y > 0 \) but only one of \( x \) or \( y \) may be zero.

The function is a degenerate special case of the integral of the third kind

\[
\frac{1}{
\int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)(t+(\rho))}}
}
\]

with \( \rho \neq 0 \), \( x, y, z \geq 0 \) with at most one equality holding. Thus \( R(x,y,z) = R(x,y,z,z) \). The normalisation of both these functions
D J

is chosen so that

\[ R(x, x, x, x) = \frac{1}{x/x} \]

The algorithms used for all these functions are based on duplication theorems. These allow a recursion system to be established which constructs a new set of arguments from the old using a combination of arithmetic and geometric means. The value of the function at the original arguments can then be simply related to the value at the new arguments. These recursive reductions are used until the arguments differ from the mean by an amount small enough for a Taylor series about the mean to give sufficient accuracy when retaining terms of order less than six. Each step of the recurrences reduces the difference from the mean by a factor of four, and as the truncation error is of order six, the truncation error goes like \((4096)^{-n}\), where \(n\) is the number of iterations.

The above forms can be related to the more traditional canonical forms (see Abramowitz and Stegun [1], 17.2).

If we write \(q = \cos(\phi), r = 1 - m \sin(\phi), s = 1 + n \sin(\phi)\), where \(0 < \phi \leq -(\pi)\), we have: the elliptic integral of the first kind:

\[
F(\phi|m) = \int_0^1 \left( 1 - t^2 \right) \left( 1 - mt^2 \right) \, dt = \sin(\phi) \cdot R(q, r, 1);
\]

the elliptic integral of the second kind:

\[
E(\phi|m) = \int_0^1 \left( 1 - t^2 \right) \left( 1 - mt^2 \right) \, dt
\]

\[
= \sin(\phi) \cdot R(q, r, 1) - m \sin(\phi) \cdot R(q, r, 1)
\]
the elliptic integral of the third kind:

\[
\int_0^1 \frac{1}{(1-t)(1-mt)(1+nt)} \, dt
\]

\[
= \sin(\phi) \text{R}(q,r,1) - \frac{n \sin(\phi)}{3} \text{R}(q,r,1,s)
\]

Also the complete elliptic integral of the first kind:

\[
\int_0^{\pi/2} \frac{1}{(1-m \sin(\theta))} \, d\theta = \text{R}(0,1-m,1);
\]

the complete elliptic integral of the second kind:

\[
\int_0^{\pi/2} \frac{1}{(1-m \sin(\theta))} \, d\theta = \text{R}(0,1-m,1) - m \text{R}(0,1-m,1).
\]

2.3. Bessel and Airy Functions of a Complex Argument

The routines for Bessel and Airy functions of a real argument are based on Chebyshev expansions, as described in Section 2.1. The routines for functions of a complex argument, however, use different methods. These routines relate all functions to the modified Bessel functions \( I^{(\nu)}(z) \) and \( K^{(\nu)}(z) \) computed in the right-half complex plane, including their analytic continuations. \( I^{(\nu)} \) and \( K^{(\nu)} \) are computed by different methods according to the values of \( z \) and \( \nu \). The methods include power series, asymptotic expansions and Wronskian evaluations. The relations between functions are based on well known formulae (see Abramowitz and Stegun [1]).

2.4. References

3. Recommendations on Choice and Use of Routines

3.1. Elliptic Integrals

IMPORTANT ADVICE: users who encounter elliptic integrals in the course of their work are strongly recommended to look at transforming their analysis directly to one of the Carlson forms, rather than the traditional canonical Legendre forms. In general, the extra symmetry of the Carlson forms is likely to simplify the analysis, and these symmetric forms are much more stable to calculate.

The routine S21BAF for R is largely included as an auxiliary to C
the other routines for elliptic integrals. This integral essentially calculates elementary functions, e.g.

\[
\ln x = (x-1) R \left(\frac{1+x}{2}, x\right), x > 0; \\
C(\frac{2}{2})
\]

\[
\arcsin x = x R (1-x, 1), |x| <= 1; \\
C
\]

\[
\arcsinh x = x R (1+x, 1), \text{etc} \\
C
\]

In general this method of calculating these elementary functions
is not recommended as there are usually much more efficient specific routines available in the Library. However, S21BAF may be used, for example, to compute \( \ln x/(x-1) \) when \( x \) is close to 1, without the loss of significant figures that occurs when \( \ln x \) and \( x-1 \) are computed separately.

3.2. Bessel and Airy Functions

For computing the Bessel functions \( J^{(\nu)}(x) \), \( Y^{(\nu)}(x) \), \( I^{(\nu)}(x) \) and \( K^{(\nu)}(x) \) where \( x \) is real and \( \nu = 0 \) or 1, special routines are provided, which are much faster than the more general routines that allow a complex argument and arbitrary real \( \nu \geq 0 \) functions and their derivatives \( \text{Ai}(x), \text{Bi}(x), \text{Ai}'(x), \text{Bi}'(x) \) for a real argument which are much faster than the routines for complex arguments.

3.3. Index

Airy function, \( \text{Ai} \), real argument S17AGF
Airy function, \( \text{Ai}' \), real argument S17AJF
Airy function, \( \text{Ai} \) or \( \text{Ai}' \), complex argument, optionally scaled S17DGF
Airy function, \( \text{Bi} \), real argument S17AHF
Airy function, \( \text{Bi}' \), real argument S17AKF
Airy function, \( \text{Bi} \) or \( \text{Bi}' \), complex argument, optionally scaled S17DHF
Bessel function, \( J^{0}(x) \), real argument S17AEF
Bessel function, \( J^{1}(x) \), real argument S17AFF
Bessel function, \( J^{(\nu)}(x) \), complex argument, optionally scaled S17DEF
Bessel function, \( Y^{0}(x) \), real argument S17ACF
Bessel function, \( Y^{1}(x) \), real argument S17ADF
Bessel function, \( Y^{(\nu)}(x) \), complex argument, optionally scaled S17DCF
Complement of the Error function S15ADF
Cosine Integral S13ACF
Elliptic integral, symmetrised, degenerate of 1st kind, R C S21BAF
Elliptic integral, symmetrised, of 1st kind, R F S21BBF
Elliptic integral, symmetrised, of 2nd kind, R S21BCF
Elliptic integral, symmetrised, of 3rd kind, R

D

Erf, real argument

J

Erfc, real argument

Error function

Exponential, complex

Exponential Integral

Fresnel Integral, C

Fresnel Integral, S

Gamma function

Gamma function, incomplete

Generalized Factorial function

(1) (2)

Hankel function $H_\nu$, complex argument, optionally scaled

Incomplete Gamma function

Jacobian elliptic functions, $sn$, $cn$, $dn$

Kelvin function, bei $x$

Kelvin function, ber $x$

Kelvin function, kei $x$

Kelvin function, ker $x$

Logarithm of Gamma function

Modified Bessel function, $I_0$, real argument

$0$

Modified Bessel function, $I_1$, real argument

$1$

Modified Bessel function, $I_\nu$, complex argument, optionally scaled

Modified Bessel function, $K_0$, real argument

$0$

Modified Bessel function, $K_1$, real argument

$1$

Modified Bessel function, $K_\nu$, complex argument, optionally scaled

Sine integral

S -- Approximations of Special Functions

Chapter S

S01EAF Complex exponential, $e^z$

S13AAF Exponential integral $E_1(x)$
1

S13ACF  Cosine integral $\text{Ci}(x)$
S13ADF  Sine integral $\text{Si}(x)$
S14AAF  Gamma function
S14ABF  Log Gamma function
S14BAF  Incomplete gamma functions $P(a,x)$ and $Q(a,x)$
S15ADF  Complement of error function $\text{erfc} \; x$
S15AEF  Error function $\text{erf} \; x$
S17ACF  Bessel function $Y_0(x)$
S17ADF  Bessel function $Y_1(x)$
S17AEF  Bessel function $J_0(x)$
S17AFF  Bessel function $J_1(x)$
S17AGF  Airy function $\text{Ai}(x)$
S17AHF  Airy function $\text{Bi}(x)$
S17AJF  Airy function $\text{Ai}'(x)$
S17AKF  Airy function $\text{Bi}'(x)$
S17DCF  Bessel functions $Y_{\nu+a}(z)$, real $a \geq 0$, complex $z$, $(\nu)=0,1,2,\ldots$
S17DEF  Bessel functions $J_{\nu+a}(z)$, real $a \geq 0$, complex $z$, $(\nu)=0,1,2,\ldots$
S17DGF  Airy functions $\text{Ai}(z)$ and $\text{Ai}'(z)$, complex $z$
S17DHF  Airy functions $\text{Bi}(z)$ and $\text{Bi}'(z)$, complex $z$
S17DLF  Hankel functions $H_{\nu+a}(z)$, $j=1,2$, real $a \geq 0$, complex $z$,
\[(\nu)+a\]
\((\nu)=0,1,2,...\)

S18ACF Modified Bessel function \(K(x)\)

S18ADF Modified Bessel function \(K(x)\)

S18AEF Modified Bessel function \(I(x)\)

S18AFF Modified Bessel function \(I(x)\)

S18DCF Modified Bessel functions \(K(z)\), real \(a \geq 0\), complex \((\nu)+a\)

S18DEF Modified Bessel functions \(I(z)\), real \(a \geq 0\), complex \((\nu)+a\)

S19AAF Kelvin function \(ber x\)

S19ABF Kelvin function \(bei x\)

S19ACF Kelvin function \(ker x\)

S19ADF Kelvin function \(kei x\)

S20ACF Fresnel integral \(S(x)\)

S20ADF Fresnel integral \(C(x)\)

S21BAF Degenerate symmetrised elliptic integral of 1st kind \(R(x,y)\)

S21BBF Symmetrised elliptic integral of 1st kind \(R(x,y,z)\)

S21BCF Symmetrised elliptic integral of 2nd kind \(R(x,y,z)\)

S21BDF Symmetrised elliptic integral of 3rd kind \(R(x,y,z)\)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S01 -- Approximations of Special Functions

S01EAF
S01EAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S01EAF evaluates the exponential function $e^z$, for complex $z$.

2. Specification

```fortran
COMPLEX(KIND(1.0D0)) FUNCTION S01EAF (Z, IFAIL)
INTEGER IFAIL
COMPLEX(KIND(1.0D0)) Z
```

3. Description

This routine evaluates the exponential function $e^z$, taking care to avoid machine overflow, and giving a warning if the result cannot be computed to more than half precision. The function is evaluated as $e^z = e^x (\cos y + i \sin y)$, where $x$ and $y$ are the real and imaginary parts respectively of $z$.

Since $\cos y$ and $\sin y$ are less than or equal to 1 in magnitude, it is possible that $e^x$ may overflow although $e^x \cos y$ or $e^x \sin y$ does not. In this case the alternative formula $\text{sign}(\cos y)e^{x+\ln|\cos y|}$ is used for the real part of the result, and $\text{sign}(\sin y)e^{x+\ln|\sin y|}$ for the imaginary part. If either part of the result still overflows, a warning is returned through parameter IFAIL.

If $\text{Im } z$ is too large, precision may be lost in the evaluation of $\sin y$ and $\cos y$. Again, a warning is returned through IFAIL.

4. References

None.

5. Parameters

1:  Z -- COMPLEX(KIND(1.0D0))  
   \hspace{1cm} Input
   \hspace{1cm} On entry: the argument $z$ of the function.
2:  IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
  The real part of the result overflows, and is set to the
  largest safe number with the correct sign. The imaginary
  part of the result is meaningful.

IFAIL= 2
  The imaginary part of the result overflows, and is set to
  the largest safe number with the correct sign. The real part
  of the result is meaningful.

IFAIL= 3
  Both real and imaginary parts of the result overflow, and
  are set to the largest safe number with the correct signs.

IFAIL= 4
  The computed result is accurate to less than half precision,
  due to the size of Im z.

IFAIL= 5
  The computed result has no precision, due to the size of Im
  z, and is set to zero.

7. Accuracy

Accuracy is limited in general only by the accuracy of the
Fortran intrinsic functions in the computation of siny, cosy and
x
\( e \), where x=Re z, y=Im z. As y gets larger, precision will
probably be lost due to argument reduction in the evaluation of
the sine and cosine functions, until the warning error IFAIL = 4
occurs when y gets larger than \( \sqrt{1/(\text{epsilon})} \), where (epsilon) is
the machine precision. Note that on some machines, the intrinsic
functions SIN and COS will not operate on arguments larger than
about $\sqrt{1}/(\epsilon)$, and so IFAIL can never return as 4.

In the comparatively rare event that the result is computed by
\[ x + \ln|\cos y| \quad \text{and} \quad x + \ln|\sin y| \]
the formulae sign(cos $y$) $e^{x}$ and sign(sin $y$) $e^{x}$, a
further small loss of accuracy may be expected due to rounding
errors in the logarithmic function.

8. Further Comments

None.

9. Example

The example program reads values of the argument $z$ from a file,
evaluates the function at each value of $z$ and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S13 -- Approximations of Special Functions
S13AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S13AAF returns the value of the exponential integral $E_1(x)$, via

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S13AAF (X, IFAIL)} \\
\text{INTEGER IFAIL} \\
\text{DOUBLE PRECISION X}
\]

3. Description

The routine calculates an approximate value for

\[
E(x) = \int_1^{\infty} e^{-u} \, du, \quad x > 0.
\]
For $0 < x \leq 4$, the approximation is based on the Chebyshev expansion

$$ E(x) = y(t) - \ln x = a_0 T_0(t) - \ln x, \text{ where } t = -x - 1. $$

For $x > 4$,

$$ e^{-x} e^{-x} = E(x) = \frac{1}{x} \left( -y(t) = \frac{1}{x} a_0 T_0(t), \right. $$

where $t = \frac{-1.0 + 14.5/(x + 3.25)}{3.25 + x}$

In both cases, $-1 \leq t \leq +1$.

To guard against producing underflows, if $x > x_{\text{hi}}$ the result is set directly to zero. For the value $x_{\text{hi}}$, see the Users' Note for your implementation.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER Input/Output
   Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL = 1
The routine has been called with an argument less than or
equal to zero for which the function is not defined. The
result returned is zero.

7. Accuracy

If (delta) and (epsilon) are the relative errors in argument and
result respectively, then in principle,

\[
\frac{|-x|}{E(x)} \approx \frac{|(epsilon)|}{|e|} \ast (delta)
\]

so the relative error in the argument is amplified in the result
by at least a factor \( \frac{e}{E(x)} \). The equality should hold if
\( 1 \) (delta) is greater than the machine precision ((delta) due to
data errors etc) but if (delta) is simply a result of round-off
in the machine representation, it is possible that an extra
figure may be lost in internal calculation and round-off.

The behaviour of this amplification factor is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

It should be noted that, for small \( x \), the amplification factor
tends to zero and eventually the error in the result will be
limited by machine precision.

For large \( x \),

\[(epsilon)^x(\delta)=(\Delta),\]

the absolute error in the argument.

8. Further Comments

None.

9. Example
The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S13 -- Approximations of Special Functions
S13ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S13ACF returns the value of the cosine integral

\[ \text{Ci}(x) = (\gamma) + \ln x + \int_{0}^{x} \frac{1}{\cos u - 1} \, du, \quad x > 0 \]

via the routine name, where (\gamma) denotes Euler’s constant.

2. Specification

```FORTRAN
DOUBLE PRECISION FUNCTION S13ACF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for \text{Ci}(x).

For \(0 < x \leq 16\) it is based on the Chebyshev expansion

\[ \text{Ci}(x) = \ln x + \sum_{r=0}^{16} \frac{a_r}{t^{2r+1}} , \quad t = 2^{(2^{-r} - 1)} \cdot x \]

For \(16 < x < \infty\) where the value of \(x\) is given in the Users’ Note

\( hi \) for your implementation,
\[ Ci(x) = \frac{f(x) \sin x}{x} - \frac{g(x) \cos x}{x^2} \]

where \( f(x) = \sum_{r=0}^{\infty} f_r T(t) \) and \( g(x) = \sum_{r=0}^{\infty} g_r T(t), \quad t = 2(\text{--) } -1 \).

For \( x \geq x_0 \), \( Ci(x) = 0 \) to within the accuracy possible (see Section 7).

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION
   On entry: the argument \( x \) of the function. Constraint: \( X > 0 \).

2: \( IFAIL \) -- INTEGER
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The routine has been called with an argument less than or equal to zero for which the function is not defined. The result returned is zero.

7. Accuracy

If \( E \) and \( \epsilon \) are the absolute and relative errors in the result and \( \delta \) is the relative error in the argument then in principle these are related by

\[ |(\delta)\cos x| \]
\[ |E| = |(\delta)\cos x| \quad \text{and} \quad |(\epsilon)| = |-------|.
\]
That is accuracy will be limited by machine precision near the origin and near the zeros of \(\cos x\), but near the zeros of \(\text{Ci}(x)\) only absolute accuracy can be maintained.

The behaviour of this amplification is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

\[
\sin x
\]

For large values of \(x\), \(\text{Ci}(x)\sim \frac{\sin x}{x}\) therefore

\[
\varepsilon \sim (\delta) x \cot x
\]

and since \((\delta)\) is limited by the finite precision of the machine it becomes impossible to return results which have any relative accuracy. That is, when \(x \geq 1/(\delta)\) we have that \(|\text{Ci}(x)| \leq 1/x^E\) and hence is not significantly different from zero.

Hence \(x\) is chosen such that for values of \(x \geq x^\text{hi}\), \(\text{Ci}(x)\) in principle would have values less than the machine precision and so is essentially zero.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument \(x\) from a file, evaluates the function at each value of \(x\) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

S13ADF returns the value of the sine integral

\[ \frac{x}{\sin u} \int_0^u \frac{\sin t}{t} \, dt \]

via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S13ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for \( \text{Si}(x) \).

For \(|x| \leq 16.0\) it is based on the Chebyshev expansion

\[ \text{Si}(x) = x > a_\infty T_n(x), \quad n = 2(\infty) - 1. \]

For \(16 < |x| < x_h\), where \(x_h\) is an implementation dependent number,

\[ \text{Si}(x) = \text{sign}(x) \left\{ \begin{array}{c} f(x) \cos x \quad g(x) \sin x \\end{array} \right\} \]

where \(f(x) = > a_\infty T_n(t), \quad g(x) = > g_\infty T_n(t), \quad n = 2(\infty) - 1.\)

For \(|x| \geq x_h\), \( \text{Si}(x) = -(\pi) \text{sign} x \) to within machine precision.

4. References

5. Parameters

1: \( X \) -- DOUBLE PRECISION  
   Input  
   On entry: the argument \( x \) of the function.

2: \( IFAIL \) -- INTEGER  
   Input/Output  
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not 
   familiar with this parameter (described in the Essential 
   Introduction) the recommended value is 0.

   On exit: \( IFAIL = 0 \) unless the routine detects an error (see 
   Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter \( IFAIL \) 
has been included for consistency with other routines in this 
chapter.

7. Accuracy

If \( (\delta) \) and \( (\epsilon) \) are the relative errors in the argument 
and result, respectively, then in principle

\[
\frac{|(\delta)\sin x|}{|(\epsilon)|} \approx \frac{|(\delta)|}{|\text{Si}(x)|}.
\]

The equality may hold if \( (\delta) \) is greater than the machine 
precision \( (\delta) \) due to data errors etc) but if \( (\delta) \) is 
simply due to round-off in the machine representation, then since 
the factor relating \( (\delta) \) to \( (\epsilon) \) is always less than one, 
the accuracy will be limited by machine precision.

8. Further Comments

For details of the time taken by the routine see the Users’ Note 
for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, 
evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for 
all example programs is distributed with the NAG Foundation 
Library software and should be available on-line.
CHAPTER 15. CHAPTER N

S14 -- Approximations of Special Functions

S14AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S14AAF returns the value of the Gamma function \( \Gamma(x) \), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S14AAF (X, IFAIL)}
\]

\[
\begin{align*}
\text{INTEGER} & \quad \text{IFAIL} \\
\text{DOUBLE PRECISION} & \quad \text{X}
\end{align*}
\]

3. Description

This routine evaluates an approximation to the Gamma function \( \Gamma(x) \). The routine is based on the Chebyshev expansion:

\[
\Gamma(1+u) = \sum_{r=0}^{\infty} a_r T_r(t), \quad 0 \leq u < 1, \quad t = 2u - 1,
\]

and uses the property \( \Gamma(x+1) = x \Gamma(x) \). If \( x = N + 1 + u \) where \( N \) is integral and \( 0 \leq u < 1 \) then it follows that:

- for \( N > 0 \), \( \Gamma(x) = (x-1)(x-2)\ldots(x-N)\Gamma(1+u) \),
- for \( N = 0 \), \( \Gamma(x) = \Gamma(1+u) \),
- \( \Gamma(1+u) \),
- for \( N < 0 \), \( \Gamma(x) = \frac{\Gamma(1+u)}{x(x+1)(x+2)\ldots(x-N-1)} \)

There are four possible failures for this routine:

(i) if \( x \) is too large, there is a danger of overflow since \( \Gamma(x) \) could become too large to be represented in the machine;

(ii) if \( x \) is too large and negative, there is a danger of underflow;
(iii) if \( x \) is equal to a negative integer, \((\Gamma)(x)\) would overflow since it has poles at such points;

(iv) if \( x \) is too near zero, there is again the danger of overflow on some machines. For small \( x \), \((\Gamma)(x)^\sim = -\), and

\[ \frac{1}{x} \]

on some machines there exists a range of non-zero but small values of \( x \) for which \( 1/x \) is larger than the greatest representable value.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
   Input
   On entry: the argument \( x \) of the function. Constraint: \( X \) must not be a negative integer.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   The argument is too large. On soft failure the routine returns the approximate value of \((\Gamma)(x)\) at the nearest valid argument.

IFAIL= 2
   The argument is too large and negative. On soft failure the routine returns zero.

IFAIL= 3
   The argument is too close to zero. On soft failure the routine returns the approximate value of \((\Gamma)(x)\) at the nearest valid argument.

IFAIL= 4
   The argument is a negative integer, at which value
(Gamma)(x) is infinite. On soft failure the routine returns a large positive value.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and the result respectively. If (delta) is somewhat larger than the machine precision (i.e., is due to data errors etc), then (epsilon) and (delta) are approximately related by:

\[ (epsilon) \approx |x(\Psi)(x)| (delta) \]

(provided (epsilon) is also greater than the representation error). Here (Psi)(x) is the digamma function \( (Gamma)'(x) \) \( (Gamma)(x) \)

Figure 1 shows the behaviour of the error amplification factor \( |x(\Psi)(x)| \).

Figure 1
Please see figure in printed Reference Manual

If (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

There is clearly a severe, but unavoidable, loss of accuracy for arguments close to the poles of (Gamma)(x) at negative integers. However relative accuracy is preserved near the pole at x=0 right up to the point of failure arising from the danger of overflow.

Also accuracy will necessarily be lost as x becomes large since in this region

\[ (epsilon) \approx (delta)xlnx. \]

However since (Gamma)(x) increases rapidly with x, the routine must fail due to the danger of overflow before this loss of accuracy is too great. (e.g. for x=20, the amplification factor \approx 60.)

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S14 -- Approximations of Special Functions
S14ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S14ABF returns a value for the logarithm of the Gamma function, ln(Gamma)(x), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S14ABF (X, IFAIL)}
\]
\[
\text{INTEGER IFAIL}
\]
\[
\text{DOUBLE PRECISION X}
\]

3. Description

This routine evaluates an approximation to ln(Gamma)(x). It is based on two Chebyshev expansions.

For 0<x<=1, ln(Gamma)(x)=-lnx to within machine accuracy.

For x <x<=15.0, the recursive relation

\[
(\text{Gamma})(1+x)=x(\text{Gamma})(x)
\]

is used to reduce the calculation to one involving (Gamma)(1+u), 0<u<1 which is evaluated as:

\[
(\text{Gamma})(1+u) = \sum_{r=0}^{\infty} a_r T_r(t), \quad t=2u-1.
\]

Once (Gamma)(x) has been calculated, the required result is produced by taking the logarithm.

For 15.0<x=x ,

big
\begin{equation}
\ln(\Gamma(x)) = (x-\frac{1}{2})\ln x - x + \frac{\ln 2(\pi)}{2} + \frac{y(x)}{x}
\end{equation}

where $y(x) = \sum_{r=0}^{\infty} \frac{b^r T(t)}{r!}$, $t = 2(\frac{x}{2}) - 1$.

For $x < x_{\text{min}}$, the term $y(x)/x$ is negligible and so its calculation is omitted.

For $x > x_{\text{max}}$ there is a danger of setting overflow so the routine must fail.

For $x \leq 0.0$ the function is not defined and the routine fails.

Note: $x$ is calculated so that if $x < x_{\text{small}}$, $(\Gamma)(x) = 1/x$ to within machine accuracy.

$\ln(\Gamma(x))$ is close to the value returned by X02ALF(*)

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION 

On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER 

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X <= 0.0, the function is undefined. On soft failure, the routine returns zero.

IFAIL= 2
X is too large, the function would overflow. On soft failure, the routine returns the value of the function at the largest permissible argument.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively, and E be the absolute error in the result.

If (delta) is somewhat larger than the relative machine precision, then

\[ |x\Psi(x)| E \approx \frac{|x\Psi(x)|}{\ln(\Gamma(x))} |(delta)| \]

where \((\Psi)(x)\) is the digamma function \(\frac{1}{\Gamma(x)}\). Figure 1 and Figure 2 show the behaviour of these error amplification factors.

Figure 1
Please see figure in printed Reference Manual

Figure 2
Please see figure in printed Reference Manual

These show that relative error can be controlled, since except near x=1 or 2 relative error is attenuated by the function or at least is not greatly amplified.

\[
\left(1 + \frac{1}{\ln x}\right)
\]

For large x, \((epsilon)^{-1}=(1+\cdots)(delta)\) and for small x,
\[
1
\]

\((epsilon)^{-1}=\cdots(delta)\).
The function $\ln(\Gamma(x))$ has zeros at $x=1$ and $2$ and hence relative accuracy is not maintainable near those points. However absolute accuracy can still be provided near those zeros as is shown above.

If however, $(\delta)$ is of the order of the machine precision, then rounding errors in the routine’s internal arithmetic may result in errors which are slightly larger than those predicted by the equalities. It should be noted that even in areas where strong attenuation of errors is predicted the relative precision is bounded by the effective machine precision.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
This subroutine evaluates the incomplete gamma functions in the normalised form

\[
P(a,x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt,
\]

\[
Q(a,x) = \frac{1}{\Gamma(a)} \int_x^\infty t^{a-1} e^{-t} dt,
\]

with \(x \geq 0\) and \(a > 0\), to a user-specified accuracy. With this normalisation, \(P(a,x) + Q(a,x) = 1\).

Several methods are used to evaluate the functions depending on the arguments \(a\) and \(x\), the methods including Taylor expansion for \(P(a,x)\), Legendre's continued fraction for \(Q(a,x)\), and power series for \(Q(a,x)\). When both \(a\) and \(x\) are large, and \(a \approx x\), the uniform asymptotic expansion of Temme [3] is employed for greater efficiency - specifically, this expansion is used when \(a \geq 20\) and \(0.7a \leq x \leq 1.4a\).

Once either of \(P\) or \(Q\) is computed, the other is obtained by subtraction from 1. In order to avoid loss of relative precision in this subtraction, the smaller of \(P\) and \(Q\) is computed first.

This routine is derived from subroutine GAM in Gautschi [2].

4. References


5. Parameters

1: A -- DOUBLE PRECISION
   On entry: the argument \(a\) of the functions. Constraint: \(A > 0\)
CHAPTER 15

2: X -- DOUBLE PRECISION Input
On entry: the argument x of the functions. Constraint: X >= 0.0.

3: TOL -- DOUBLE PRECISION Input
On entry: the relative accuracy required by the user in the results. If S14BAF is entered with TOL greater than 1.0 or less than machine precision, then the value of machine precision is used instead.

4: P -- DOUBLE PRECISION Output

5: Q -- DOUBLE PRECISION Output
On exit: the values of the functions P(a,x) and Q(a,x) respectively.

6: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry A <= 0.0.

IFAIL= 2
On entry X < 0.0.

IFAIL= 3
Convergence of the Taylor series or Legendre continued fraction fails within 600 iterations. This error is extremely unlikely to occur; if it does, contact NAG.

7. Accuracy

There are rare occasions when the relative accuracy attained is somewhat less than that specified by parameter TOL. However, the error should never exceed more than one or two decimal places. Note also that there is a limit of 18 decimal places on the achievable accuracy, because constants in the routine are given to this precision.
8. Further Comments

The time taken for a call of S14BAF depends on the precision requested through TOL, and also varies slightly with the input arguments a and x.

9. Example

The example program reads values of the argument a and x from a file, evaluates the function and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S15 -- Approximations of Special Functions

S15ADF returns the value of the complementary error function, erfcx, via the routine name.

2. Specification

DOUBLE PRECISION FUNCTION S15ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

The routine calculates an approximate value for the complement of the error function

\[
\text{erfc } x = \frac{2}{\sqrt{\pi}} \int_{-x}^{\infty} e^{-u^2} \, du = 1 - \text{erf } x.
\]

For x \geq 0, it is based on the Chebyshev expansion
2
-\text{erfc} x= e^{-y(x)},

where \( y(x) = \sum_{r=0}^{\infty} a_r \frac{t^r}{r!} \) and \( t = \frac{(x-3.75)}{(x+3.75)} \), \(-1 \leq t \leq +1\).

For \( x \geq x_{\text{hi}} \), where there is a danger of setting underflow, the result is returned as zero.

2
-\text{erfc} x = 2 - e^{-y(|x|)}.

For \( x < 0 \), the result is returned as 2.0 which is correct to low within machine precision. The values of \( x \) and \( x_{\text{low}} \) are given in the Users' Note for your implementation.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
   Input
   On entry: the argument \( x \) of the function.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

If \((\text{delta})\) and \((\text{epsilon})\) are relative errors in the argument and
result, respectively, then in principle
\[
\begin{array}{|c|c|c|}
\hline
 & 2 & -x \\
\hline
\epsilon & 2\epsilon & \sqrt{\pi}\text{erfc }x \\
\hline
\end{array}
\]
\(\frac{\epsilon}{\epsilon} \approx \frac{\delta}{\delta}.
\]

That is, the relative error in the argument, \(x\), is amplified by a factor \(\frac{2}{\sqrt{\pi}\text{erfc }x}\) in the result.

\(\sqrt{\pi}\text{erfc }x\)

The behaviour of this factor is shown in Figure 1.

Figure 1

Please see figure in printed Reference Manual

It should be noted that near \(x=0\) this factor behaves as \(\frac{2x}{\sqrt{\pi}}\) and hence the accuracy is largely determined by the machine precision. Also for large negative \(x\), where the factor is \(\frac{2}{\sqrt{\pi}}\text{erfc }x\), accuracy is mainly limited by machine precision.

\(\frac{1}{\sqrt{\pi}}\)

However, for large positive \(x\), the factor becomes \(\frac{2}{\sqrt{\pi}}x\) and to an extent relative accuracy is necessarily lost. The absolute accuracy \(E\) is given by
\[
\begin{array}{|c|c|c|}
\hline
 & 2 & -x \\
\hline
\epsilon & 2\epsilon & \sqrt{\pi}\text{erfc }x \\
\hline
\end{array}
\]
\(\frac{\epsilon}{\epsilon} = \frac{\delta}{\delta}.
\]
so absolute accuracy is guaranteed for all $x$.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S15 -- Approximations of Special Functions

S15AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S15AEF returns the value of the error function $\text{erfx}$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S15AEF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

Evaluates the error function,

$$
\text{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.
$$

---
For $|x| \leq 2$,
\[
\text{erf } x = x > a \ T (t), \text{ where } t = -x - 1.
\]
\[
r = 0
\]

For $2 < |x| < x$,
\[
\text{erf } x = \text{sign}(x) \left\{ \frac{1}{2} \right\} - x^7
\]
\[
\text{erf } x = \text{sign}(x) \left\{ 1 - \frac{1}{2} \right\} - x^3
\]
\[
\text{erf } x = \text{sign}(x) \left\{ \frac{1}{\pi} \right\} r = 0
\]

For $|x| \geq x$,
\[
\text{erf } x = \text{sign}(x)
\]

$x$ is the value above which erf $x = \pm 1$ within machine precision.

Its value is given in the Users' Note for your implementation.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   Input On entry: the argument $x$ of the function.

2: IFAIL -- INTEGER
   Input/Output On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy
On a machine with approximately 11 significant figures the routine agrees with available tables to 10 figures and consistency checking with S15ADF of the relation

\[ \text{erf } x + \text{erfc } x = 1.0 \]

shows errors in at worst the 11th figure.

8. Further Comments

None.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions

S17ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17ACF returns the value of the Bessel Function \( Y(x) \), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S17ACF (X, IFAIL)}
\]

\[
\text{INTEGER IFAIL}
\]

\[
\text{DOUBLE PRECISION X}
\]

3. Description

This routine evaluates an approximation to the Bessel Function of the second kind \( Y(x) \).

Note: \( Y(x) \) is undefined for \( x \leq 0 \) and the routine will fail for
such arguments.

The routine is based on four Chebyshev expansions:

For 0 < x ≤ 8,

\[
Y(x) = \frac{-\ln(x)}{x} + \sum_{r=0}^{\infty} a_r T_r(t) + b_r T_r(t), \text{ with } t = 2^r - 1,
\]

where \( a_r \) and \( b_r \) are constants.

For x > 8,

\[
Y(x) = \frac{\sin(x) - \cos(x)}{x} + \sum_{r=0}^{\infty} c_r T_r(t) + d_r T_r(t), \text{ with } t = 2^r - 1.
\]

For x near zero, \( Y(x) \approx \frac{-\ln(x) + (\gamma)}{x} \), where \( (\gamma) \) denotes Euler's constant. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For very large x, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of \( Y(x) \); only the amplitude, \( \frac{1}{x} \), can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision: the routine will fail if x > ~1/machine precision (see the Users' Note for your implementation for details).
4. References


5. Parameters

1: X -- DOUBLE PRECISION
   Input
   On entry: the argument x of the function. Constraint: X > 0.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   X is too large. On soft failure the routine returns the amplitude of the Y oscillation, 1/(2/(pi)x).

IFAIL= 2
   X <= 0.0, Y is undefined. On soft failure the routine returns zero.

7. Accuracy

Let (delta) be the relative error in the argument and E be the absolute error in the result. (Since Y(x) oscillates about zero, absolute error and not relative error is significant, except for very small x.)

If (delta) is somewhat larger than the machine representation error (e.g. if (delta) is due to data errors etc), then E and (delta) are approximately related by

\[ E \approx |xY(x)| \times (\delta) \]
(provided E is also within machine bounds). Figure 1 displays the
behaviour of the amplification factor \(|xY(x)|\).

Figure 1
Please see figure in printed Reference Manual

However, if \((\delta)\) is of the same order as the machine
representation errors, then rounding errors could make E slightly
larger than the above relation predicts.

For very small \(x\), the errors are essentially independent of
\((\delta)\) and the routine should provide relative accuracy bounded
by the machine precision.

For very large \(x\), the above relation ceases to apply. In this

\[
\frac{2}{\pi} \left( \frac{\pi}{4} \right) \sin\left( x - \frac{\pi}{4} \right)
\]

region, \(Y(x)\) = \(-\frac{2}{\pi} \sin(x - \frac{\pi}{4})\). The amplitude \(\frac{2}{\pi} \sin(x - \frac{\pi}{4})\) can be calculated with reasonable accuracy for all \(x\), but
\[
\sin(x - \frac{\pi}{4})
\]

cannot. If \(x - \frac{\pi}{4}\) is written as \(2N\pi + (\theta)\)
\[
\sin(x - \frac{\pi}{4})
\]

where \(N\) is an integer and \(0 \leq (\theta) < 2\pi\), then \(\sin(x - \frac{\pi}{4})\) is
determined by \((\theta)\) only. If \(x > \frac{\pi}{2}\), \((\theta)\) cannot be
determined with any accuracy at all. Thus if \(x\) is greater than,
or of the order of the inverse of machine precision, it is
impossible to calculate the phase of \(Y(x)\) and the routine must
fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note
for your implementation.

9. Example

The example program reads values of the argument \(x\) from a file,
evaluates the function at each value of \(x\) and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation.
Library software and should be available on-line.

S17 -- Approximations of Special Functions
S17ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17ADF returns the value of the Bessel Function $Y_1(x)$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S17ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Bessel Function of the second kind $Y_1(x)$.

Note: $Y_1(x)$ is undefined for $x\leq 0$ and the routine will fail for such arguments.

The routine is based on four Chebyshev expansions:

For $0<x\leq 8$,

$$Y_1(x) = \frac{\ln x}{8} \sum_{r=0}^{2} \left[ a_r T_r(t) - b_r T_r(t) \right],$$

where $a_r = \frac{2}{(\pi)^{r+1}}$ and $b_r = \frac{2}{(\pi)^{r+1}}$.

For $x>8$,
\[
Y(x) = \frac{1}{\pi x} \left\{ \frac{1}{(\pi)} \frac{1}{(4)} \frac{1}{1} \right\} \left\{ P(x)\sin(x-3) + Q(x)\cos(x-3) \right\}
\]

where
\[
P(x) = \sum_{r=0}^{2} c_r T(t),
\]
and
\[
Q(x) = \sum_{r=0}^{2} d_r T(t), \text{ with } t = 2^{(x)} - 1.
\]

For \(x\) near zero, \(Y(x) \approx -\frac{1}{\pi x}\). This approximation is used when \(x\) is sufficiently small for the result to be correct to machine precision. For extremely small \(x\), there is a danger of overflow in calculating \(-\frac{1}{\pi x}\) and for such arguments the routine will fail.

For very large \(x\), it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of \(Y(x)\), only the amplitude, \(-\frac{1}{\pi x}\), can be determined and this is returned on soft failure. The range for which this occurs is roughly related to machine precision; the routine will fail if \(x > \frac{1}{\text{machine precision}}\) (see the Users' Note for your implementation for details).

4. References


5. Parameters

1: \(X\) -- DOUBLE PRECISION  
   Input  
   On entry: the argument \(x\) of the function. Constraint: \(X > 0\).
2: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).  

6. Error Indicators and Warnings  

Errors detected by the routine:  

IFAIL= 1  
   X is too large. On soft failure the routine returns the  
   /  2  
   amplitude of the Y oscillation, / -------.  
   1 \  \ (pi)x  

IFAIL= 2  
   X <= 0.0, Y is undefined. On soft failure the routine  
   1  
   returns zero.  

IFAIL= 3  
   X is too close to zero, there is a danger of overflow. On  
   soft failure, the routine returns the value of Y (x) at the  
   1  
   smallest valid argument.  

7. Accuracy  

Let (delta) be the relative error in the argument and E be the  
absolute error in the result. (Since Y (x) oscillates about zero,  
absolute error and not relative error is significant, except for  
very small x.)  

If (delta) is somewhat larger than the machine precision (e.g. if  
(delta) is due to data errors etc), then E and (delta) are  
approximately related by:  

E^* = |xY (x)-Y (x)|/(delta)  
    0  1  

(provided E is also within machine bounds). Figure 1 displays the  
behaviour of the amplification factor |xY (x)-Y (x)|.
However, if (delta) is of the same order as machine precision, then rounding errors could make E slightly larger than the above relation predicts.

For very small x, absolute error becomes large, but the relative error in the result is of the same order as (delta).

For very large x, the above relation ceases to apply. In this region, Y(x)~\frac{2\pi}{x} - \sin(\pi x)\div x. The amplitude \sin(\pi x)\div x can be calculated with reasonable accuracy for all x, but \sin(\pi x)\div x cannot. If x=\frac{2n\pi}{4}+\theta where n is an integer and 0\leq\theta<2\pi, then \sin(\pi x)\div x is determined by \theta only. If x>(delta), \theta cannot be determined with any accuracy at all. Thus if x is greater than, or of the order of, the inverse of the machine precision, it is impossible to calculate the phase of Y(x) and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AEF returns the value of the Bessel Function \( J_0(x) \), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S17AEF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Bessel Function of the first kind \( J_0(x) \).

Note: \( J_0(-x)=J_0(x) \), so the approximation need only consider \( x \geq 0 \).

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 8 \),

\[
J_0(x) = \sum_{r=0}^{2} a_r T_r(t), \quad \text{with } t = 2(-x)^{-1} - 1.
\]

For \( x > 8 \),

\[
J_0(x) = \frac{2}{\pi x} \left\{ \frac{(\pi)}{4} P_0(x) \cos(x - \frac{\pi}{4}) - Q_0(x) \sin(x - \frac{\pi}{4}) \right\},
\]

where \( P_0(x) = \sum_{r=0}^{2} b_r T_r(t), \quad \text{with } t = 2(-x)^{-1} - 1 \),

and \( Q_0(x) = \sum_{r=0}^{2} c_r T_r(t), \quad \text{with } t = 2(-x)^{-1} - 1 \).
For $x$ near zero, $J_0(x) \approx 1$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.

For very large $x$, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $J_0(x)$; only the amplitude, $\sqrt{\frac{2}{\pi|x|}}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if $|x|>\frac{1}{\text{machine precision}}$ (see the Users' Note for your implementation).

4. References


5. Parameters

1: $X \rightarrow$ DOUBLE PRECISION
   Input
   On entry: the argument $x$ of the function.

2: IFAIL $\rightarrow$ INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL$= 1$
   $X$ is too large. On soft failure the routine returns the
amplitude of the \( J \) oscillation, \( \sqrt{\frac{2}{\pi|x|}} \)

7. Accuracy

Let \( \delta \) be the relative error in the argument and \( E \) be the absolute error in the result. (Since \( J(x) \) oscillates about zero, absolute error and not relative error is significant.)

If \( \delta \) is somewhat larger than the machine precision (e.g. if \( \delta \) is due to data errors etc), then \( E \) and \( \delta \) are approximately related by:

\[
E = |xJ(x)|\delta
\]

(provided \( E \) is also within machine bounds). Figure 1 displays the behaviour of the amplification factor \( |xJ(x)| \).

---

Figure 1

Please see figure in printed Reference Manual

However, if \( \delta \) is of the same order as machine precision, then rounding errors could make \( E \) slightly larger than the above relation predicts.

For very large \( x \), the above relation ceases to apply. In this region, \( J(x) \sim \sqrt{\frac{2}{\pi|x|}} \cos(x - \frac{\pi}{4}) \). The amplitude \( \sqrt{\frac{2}{\pi|x|}} \) can be calculated with reasonable accuracy for all \( x \) but \( \cos(x - \frac{\pi}{4}) \) cannot. If \( x - \frac{\pi}{4} \) is written as \( \frac{2N\pi + \theta}{4} \) where \( N \) is an integer and \( 0 \leq \theta < 2\pi \), then \( \cos(x - \frac{\pi}{4}) \) is determined by \( \theta \) only. If \( x > \frac{\pi}{2}(\delta) \), \( \theta \) cannot be determined with any accuracy at all. Thus if \( x \) is greater than, or of the order of, the inverse of the machine
precision, it is impossible to calculate the phase of \( J_0(x) \) and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions
S17AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AFF returns the value of the Bessel Function \( J_1(x) \), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S17AFF} \ (X, \ IFAIL) \\
\text{INTEGER} \quad \text{IFAIL} \\
\text{DOUBLE PRECISION} \quad X
\]

3. Description

This routine evaluates an approximation to the Bessel Function of the first kind \( J_1(x) \).

Note: \( J_{-1}(x) = -J_1(x) \), so the approximation need only consider \( x \geq 0 \).

The routine is based on three Chebyshev expansions:
For $0 < x \leq 8$,

$$J(x) = - \sum_{r=0}^{2} a \ T(t), \text{ with } t = 2(-) - 1.$$ 

For $x > 8$,

$$J(x) = \sum_{r=0}^{2} \frac{(3\pi)}{1 \sqrt{1 + (\frac{4}{4})}} \ \text{P}(x) \cos(x - \frac{3\pi}{4}) - Q(x) \sin(x - \frac{3\pi}{4}).$$

where $P(x) = \sum_{r=0}^{2} b \ T(t),$ 

and $Q(x) = \sum_{r=0}^{2} c \ T(t), \text{ with } t = 2(-) - 1.$

For $x$ near zero, $J(x) \approx -$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.

For very large $x$, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $J(x)$; only the amplitude, $\sqrt{\pi|x|}$ can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if $|x| > 1/$machine precision (see the Users’ Note for your implementation for details).

4. References


5. Parameters

1: X -- DOUBLE PRECISION  
   Input
   On entry: the argument x of the function.

2: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   X is too large. On soft failure the routine returns the

   / 2
   amplitude of the J oscillation, / --------.
   1 \( \sqrt{\pi|x|} \)

7. Accuracy

Let (delta) be the relative error in the argument and E be the
absolute error in the result. (Since \( J(x) \) oscillates about zero,
absolute error and not relative error is significant.)

If (delta) is somewhat larger than machine precision (e.g. if
(delta) is due to data errors etc), then E and (delta) are
approximately related by:

\[
E = |xJ(x)-J(x)| (\text{delta})
\]

(provided E is also within machine bounds). Figure 1 displays the
behaviour of the amplification factor \(|xJ(x)-J(x)|\).

![Figure 1](image)

However, if (delta) is of the same order as machine precision,
then rounding errors could make E slightly larger than the above
relation predicts.

For very large \( x \), the above relation ceases to apply. In this

\[
\frac{2}{(3\pi)} \cos(x - \frac{3\pi}{4})
\]

region, \( J(x) \approx \frac{\sqrt{(\pi|x|)}}{2} \cdot \cos(x - \frac{3\pi}{4}) \). The amplitude

\[
\frac{1}{\sqrt{(\pi|x|)}} \cdot \frac{(4)}{3\pi}
\]

can be calculated with reasonable accuracy for all \( x \)

\[
\frac{1}{\sqrt{(\pi|x|)}} \cdot \frac{(3\pi)}{4}
\]

but \( \cos(x - \frac{3\pi}{4}) \) cannot. If \( x - \frac{3\pi}{4} \) is written as

\[
2N\pi + \theta\]

where \( N \) is an integer and \( 0 \leq \theta < 2\pi \), then

\[
\cos(x - \frac{3\pi}{4})
\]

is determined by \( \theta \) only. If \( x > \delta \),

\[
\frac{1}{4}
\]

(\( \theta \)) cannot be determined with any accuracy at all. Thus if \( x \)

is greater than, or of the order of, machine precision, it is

impossible to calculate the phase of \( J(x) \) and the routine must

fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note

for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file,

evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for

all example programs is distributed with the NAG Foundation

Library software and should be available on-line.
1. Purpose

S17AGF returns a value for the Airy function, Ai(x), via the routine name.

2. Specification

    DOUBLE PRECISION FUNCTION S17AGF (X, IFAIL)
    INTEGER IFAIL
    DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Airy function, Ai(x). It is based on a number of Chebyshev expansions:

For \( x < -5 \),

\[
Ai(x) = \frac{a(t) \sin z - b(t) \cos z}{1/4 \left( \pi \right)^{2/3} (-x)^{5/3}}
\]

where \( z = \frac{\pi}{4} - \sqrt{-x} \), and \( a(t) \) and \( b(t) \) are expansions in the variable \( t = -2(\frac{-x}{5})^{-1} \).

For \(-5 \leq x \leq 0\),

\[
Ai(x) = f(t) - xg(t),
\]

where \( f \) and \( g \) are expansions in \( t = -2(\frac{-x}{5})^{-1} \).

For \( 0 < x < 4.5 \),

\[
Ai(x) = e^{\frac{-3x}{2}} y(t),
\]

where \( y \) is an expansion in \( t = 4x/9 - 1 \).

For \( 4.5 \leq x < 9 \),
\[ Ai(x) = e^{-\frac{5x}{2}}u(t), \]

where \( u \) is an expansion in \( t = 4x/9 - 3 \).

For \( x \geq 9 \),

\[ Ai(x) = \frac{-z}{e^{v(t)}} \frac{1}{\sqrt{x}}, \]

where \( z = \sqrt{-x} \) and \( v \) is an expansion in \( t = 2\left(\frac{-z}{3}\right) - 1 \).

For \(|x| < \text{the machine precision}\), the result is set directly to \( Ai(0) \). This both saves time and guards against underflow in intermediate calculations.

For large negative arguments, it becomes impossible to calculate the phase of the oscillatory function with any precision and so the routine must fail. This occurs if \( x < -\left(\frac{3}{2\text{(epsilon)}}\right) \), where \((\text{epsilon})\) is the machine precision.

For large positive arguments, where \( Ai \) decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION
   Input
   On entry: the argument \( x \) of the function.

2: \( IFAIL \) -- INTEGER
   Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL = 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential-like and here relative error is appropriate. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

$$\frac{| xA_1'(x)|}{| A_1(x) |} = \frac{| xA_1'(x)|}{| A_1(x) |} (\delta) \leq \epsilon \approx \frac{1}{\pi^{\frac{2}{3}}} \left( \frac{5}{4} \right)^{\frac{2}{3}}$$

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, errors are strongly damped by the function and hence will be bounded by the machine precision.

For moderate negative x, the error behaviour is oscillatory but the amplitude of the error grows like

$$\text{amplitude} \approx \left( \frac{\delta}{\pi} \right)^{\frac{2}{3}}$$

However the phase error will be growing roughly like

$$\frac{\pi}{\sqrt{|x|}}$$

and hence all accuracy will be lost for large negative arguments.
due to the impossibility of calculating \( \sin \) and \( \cos \) to any

\[
\frac{2}{3} 1
\]

accuracy if \( -\sqrt{\mid x \mid} > \frac{\text{--------}}{3} \) (delta)

For large positive arguments, the relative error amplification is considerable:

\[
\frac{\text{(epsilon)}}{3} \frac{\text{---------}}{\sqrt{\mid x \mid}} (\text{delta})
\]

This means a loss of roughly two decimal places accuracy for arguments in the region of 20. However very large arguments are not possible due to the danger of setting underflow and so the errors are limited in practice.

8. Further Comments

None.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
DOUBLE PRECISION FUNCTION S17AHF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Airy function Bi(x). It is based on a number of Chebyshev expansions.

For x<-5,

\[
\frac{a(t) \cos z + b(t) \sin z}{1/4 (-x)^{2/3}}
\]

where \( z = \frac{\pi}{4} - \sqrt{-x} \) and \( a(t) \) and \( b(t) \) are expansions in the variable \( t = -2(-)^{-1} 5(x) \).

For -5<=x<=0,

\[
Bi(x) = \sqrt{3} (f(t) + x g(t)),
\]

where \( f \) and \( g \) are expansions in \( t = -2(-)^{-1} 5(x) \).

For 0<x<4.5,

\[
11x/8
Bi(x) = e^{y(t)},
\]

where \( y \) is an expansion in \( t = 4x/9 - 1 \).

For 4.5<=x<=9,

\[
5x/2
Bi(x) = e^{v(t)},
\]

where \( v \) is an expansion in \( t = 4x/9 - 3 \).
For $x \geq 9$,

$$
\frac{z}{e^{u(t)}}
$$

$$
\frac{1}{4},
$$

$$
\frac{2}{3} \quad \text{(18)}
$$

where $z = -\sqrt{x}$ and $u$ is an expansion in $t = 2(\frac{z}{3})^{-1}$.

For $|x| <$ the machine precision, the result is set directly to $Bi(0)$. This both saves time and avoids possible intermediate underflows.

For large negative arguments, it becomes impossible to calculate the phase of the oscillating function with any accuracy so the routine must fail. This occurs if $x < -(\frac{2}{3})$ (epsilon), where (epsilon) is the machine precision.

For large positive arguments, there is a danger of causing overflow since $Bi$ grows in an essentially exponential manner, so the routine must fail.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   Input
   On entry: the argument $x$ of the function.

2: $IFAIL$ -- INTEGER
   Input/Output
   On entry: $IFAIL$ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: $IFAIL = 0$ unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential-like and here relative error is appropriate. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

\[ |\frac{xBi'(x)}{Bi(x)}| E = |xBi'(x)| (delta), (epsilon) = \frac{|Bi(x)|}{-------} (delta). \]

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, errors are strongly damped and hence will be bounded essentially by the machine precision.

For moderate to large negative x, the error behaviour is clearly oscillatory but the amplitude of the error grows like amplitude

\[ \frac{5}{4} \]

\[ (E) |x| \]

\[ (\frac{---}{(delta)})^- \]

\[ (\frac{\sqrt{x/\pi}}{3})^2 \]

However the phase error will be growing roughly as \( -\sqrt{|x|} \) and hence all accuracy will be lost for large negative arguments.

This is due to the impossibility of calculating sin and cos to
For large positive arguments, the relative error amplification is considerable:

\[
\left(\frac{\text{epsilon}}{\text{delta}}\right) / 3 \left(\sqrt{x} - \sqrt{x + \text{delta}}\right).
\]

This means a loss of roughly two decimal places accuracy for arguments in the region of 20. However very large arguments are not possible due to the danger of causing overflow and errors are therefore limited in practice.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions
S17AJF
S17AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AJF returns a value of the derivative of the Airy function \( \text{Ai}(x) \), via the routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S17AJF (X, IFAIL)
```
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the derivative of the Airy function Ai(x). It is based on a number of Chebyshev expansions.

For x<-5,

\[ Ai'(x) = -x [a(t) \cos z + \frac{b(t)}{(zeta)} \sin z], \]

where \( z = \frac{2}{3} \pi (zeta), \) \( (zeta) = \sqrt{-x} \) and \( a(t) \) and \( b(t) \) are expansions in variable \( t = -2(\frac{5}{x}) - 1. \)

For -5<=x<=0,

\[ Ai'(x) = x f(t) - g(t), \]

where \( f(t) \) and \( g(t) \) are expansions in \( t = -2(\frac{5}{x}) - 1. \)

For 0<x<4.5,

\[ Ai'(x) = e^{\frac{-11x}{8}} y(t), \]

where \( y(t) \) is an expansion in \( t = 4(\frac{9}{x}) - 1. \)

For 4.5<=x<9,

\[ Ai'(x) = e^{\frac{-5x}{2}} v(t), \]

where \( v(t) \) is an expansion in \( t = 4(\frac{9}{x}) - 3. \)
For $x \geq 9$,

$$4-z A_i'(x) = \sqrt{-x} e^{u(t)},$$

where $z = -\sqrt{x}$ and $u(t)$ is an expansion in $t = 2(-z)^{1/3}$. 

For $|x| < \text{the square of the machine precision}$, the result is set directly to $A_i'(0)$. This both saves time and avoids possible intermediate underflows.

For large negative arguments, it becomes impossible to calculate a result for the oscillating function with any accuracy and so the routine must fail. This occurs for $x < -(\epsilon \pi)^{4/7}$, where $(\epsilon)$ is the machine precision.

For large positive arguments, where $A_i$ decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
   On entry: the argument $x$ of the function.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
IFAIL= 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential in character and here relative error is needed. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

$$\left| \frac{2}{x Ai(x)} \right| (delta) \approx \left| x Ai(x) \right| (epsilon) \approx \left| \frac{2}{Ai'(x)} \right| (delta).$$

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, positive or negative, errors are strongly attenuated by the function and hence will be roughly bounded by the machine precision.

For moderate to large negative x, the error, like the function, is oscillatory; however the amplitude of the error grows like

$$\frac{7/4}{\sqrt{\pi}}.$$

Therefore it becomes impossible to calculate the function with any accuracy if $|x| > \frac{7/4}{\sqrt{\pi} \cdot (delta)}$.

For large positive x, the relative error amplification is
considerable:

\[ \frac{\epsilon}{3} = \sqrt[\delta]{x}. \]

However, very large arguments are not possible due to the danger of underflow. Thus in practice error amplification is limited.

8. Further Comments

None.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S17 -- Approximations of Special Functions
S17AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AKF returns a value for the derivative of the Airy function \( \text{Bi}(x) \), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S17AKF (X, IFAIL)}
\]

\[
\text{INTEGER IFAIL}
\]

\[
\text{DOUBLE PRECISION X}
\]

3. Description

This routine calculates an approximate value for the derivative of the Airy function \( \text{Bi}(x) \). It is based on a number of Chebyshev expansions.
For $x<-5$,

$$\text{Bi}'(x) = \sqrt{-x} \left[ -a(t) \sin z + \frac{b(t)}{(zeta)} \cos z \right],$$

where \( z = \frac{(pi)}{4} \), \( (zeta) = \sqrt{-x} \), and \( a(t) \) and \( b(t) \) are expansions in the variable \( t = -2(\frac{x}{3})^{-1} \).

For $-5 \leq x \leq 0$,

$$\text{Bi}'(x) = \sqrt{3} (f(t) + g(t)),$$

where \( f \) and \( g \) are expansions in \( t = -2(\frac{x}{3})^{-1} \).

For $0 < x < 4.5$,

$$\text{Bi}'(x) = e^{\frac{3x}{2}} y(t),$$

where \( y(t) \) is an expansion in \( t = 4x/9 - 1 \).

For $4.5 \leq x < 9$,

$$\text{Bi}'(x) = e^{\frac{21x}{8}} u(t),$$

where \( u(t) \) is an expansion in \( t = 4x/9 - 3 \).

For $x \geq 9$,

$$\text{Bi}'(x) = \sqrt{x} e^{\frac{z}{2}} v(t),$$

where \( z = \sqrt{x} \) and \( v(t) \) is an expansion in \( t = 2(\frac{z}{3})^{-1} \).
For $|x| < \text{the square of the machine precision}$, the result is set directly to $B'_i(0)$. This saves time and avoids possible underflows in calculation.

For large negative arguments, it becomes impossible to calculate a result for the oscillating function with any accuracy so the routine must fail. This occurs for $x < -\left( \frac{4}{\pi} \right)$, where $(\epsilon)$ is the machine precision.

For large positive arguments, where $B'_i$ grows in an essentially exponential manner, there is a danger of overflow so the routine must fail.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   
   On entry: the argument $x$ of the function.

2: $IFAIL$ -- INTEGER
   
   On entry: $IFAIL$ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   
   On exit: $IFAIL = 0$ unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

$IFAIL = 1$

$X$ is too large and positive. On soft failure the routine returns zero.

$IFAIL = 2$

$X$ is too large and negative. On soft failure the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence
absolute error is appropriate. In the positive region the function has essentially exponential behaviour and hence relative error is needed. The absolute error, $E$, and the relative error $(\epsilon)$, are related in principle to the relative error in the argument $(\delta)$, by

$$
\frac{2}{|xBi(x)|} \approx |xBi(x)|\delta \quad (\epsilon) \approx \frac{1}{|Bi'(x)|} \delta.
$$

In practice, approximate equality is the best that can be expected. When $(\delta)$, $(\epsilon)$ or $E$ is of the order of the machine precision, the errors in the result will be somewhat larger.

For small $x$, positive or negative, errors are strongly attenuated by the function and hence will effectively be bounded by the machine precision.

For moderate to large negative $x$, the error is, like the function, oscillatory. However, the amplitude of the absolute error grows like $\frac{7/4 |x|}{\sqrt{\pi}}$. Therefore it becomes impossible to calculate the function with any accuracy if $|x| > \frac{7/4 \sqrt{\pi}}{(\delta)}$.

For large positive $x$, the relative error amplification is considerable: $\frac{(\epsilon)}{3}$ not possible due to the danger of overflow. Thus in practice the actual amplification that occurs is limited.

8. Further Comments

None.

9. Example

The example program reads values of the argument $x$ from a file,
evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S17 -- Approximations of Special Functions
S17DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DCF returns a sequence of values for the Bessel functions \( Y_n(\nu) \) for complex \( z \), non-negative \( \nu \) and \( n=0,1,\ldots,N-1 \), with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DCF (FNU, Z, N, SCALE, CY, NZ, CWRK, IFAIL)
   INTEGER N, NZ, IFAIL
   DOUBLE PRECISION FNU
   COMPLEX(KIND(1.0D0)) Z, CY(N), CWRK(N)
   CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the Bessel function \( Y_n(\nu) \), where \( z \) is complex, \(-\pi < \arg z \leq \pi\), and \( \nu \) is the real, non-negative order. The \( N \)-member sequence is generated for orders \( \nu \), \( \nu+1 \), \ldots, \( \nu+N-1 \). Optionally, the sequence is scaled by the factor \( e^{-|\Im z|} \).

Note: although the routine may not be called with \( \nu \) less than zero, for negative orders the formula

\[
Y_n(\nu) = Y_n(-\nu) \cos(\nu \pi) + J_n(-\nu) \sin(\nu \pi)
\]

may be used. (for the Bessel function \( J_n(\nu) \), see S17DEF).

The routine is derived from the routine CBESY in Amos [2]. It is

\[ (1) \] (2)
based on the relation \( Y(z) = \frac{H^1(z) - H^2(z)}{2i} \), where \( H^1(z) \) and \( H^2(z) \) are the Hankel functions of the first and second kinds respectively (see S17DLF).

When \( N \) is greater than 1, extra values of \( Y(z) \) are computed using recurrence relations.

For very large \(|z|\) or \((\nu)+N-1\), argument reduction will cause total loss of accuracy, and no computation is performed. For slightly smaller \(|z|\) or \((\nu)+N-1\), the computation is performed but results are accurate to less than half of machine precision. If \(|z|\) is very small, near the machine underflow threshold, or \((\nu)+N-1\) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: FNU -- DOUBLE PRECISION
   On entry: the order, \((\nu)\), of the first member of the sequence of functions. Constraint: FNU \(\geq 0\).

2: Z -- COMPLEX(KIND(1.D0))
   On entry: the argument, \(z\), of the functions. Constraint: \(Z \neq (0.0, 0.0)\).

3: N -- INTEGER
   On entry: the number, \(N\), of members required in the sequence \( Y(\nu) \), \( Y(\nu+1) \), ..., \( Y(\nu+N-1) \). Constraint: \(N \geq 1\).

4: SCALE -- CHARACTER*1
   On entry: the scaling option.

   If SCALE = 'U', the results are returned unscaled.
If $\text{SCALE} = 'S'$, the results are returned scaled by the
factor $e^{-|\text{Im}z|}$. Constraint: $\text{SCALE} = 'U'$ or 'S'.

5: \text{CY} (\text{N}) \rightarrow \text{COMPLEX(KIND(1.0D)) array Output}
On exit: the N required function values: $\text{CY}(i)$ contains
$Y^{(\nu+1-i)}(z)$, for $i=1,2,...,N$.

6: \text{NZ} \rightarrow \text{INTEGER Output}
On exit: the number of components of \text{CY} that are set to zero
due to underflow. The positions of such components in the
array \text{CY} are arbitrary.

7: \text{CWRK} (\text{N}) \rightarrow \text{COMPLEX(KIND(1.0D)) array Workspace}

8: \text{IFAIL} \rightarrow \text{INTEGER Input/Output}
On entry: \text{IFAIL} must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: \text{IFAIL} = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \text{IFAIL} = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04A AF).

IFAIL= 1
On entry $\text{FNU} < 0.0$,
        $\text{Z} = (0.0, 0.0)$,
        $\text{N} < 1$,
        or $\text{SCALE} /= 'U'$ or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of
overflow, because $\text{ABS(Z)}$ is less than a machine-dependent
threshold value (given in the Users’ Note for your
implementation).

IFAIL= 3
No computation has been performed due to the likelihood of
overflow, because $\text{FNU} + \text{N} - 1$ is too large - how large
depends on $Z$ as well as the overflow threshold of the machine.

IFAIL= 4
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S17DCF are accurate to less than half of machine precision. This error exit may occur if either ABS($Z$) or $\text{FNU} + N - 1$ is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S17DCF would be lost. This error exit may occur if either ABS($Z$) or $\text{FNU} + N - 1$ is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 6
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DCF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used $t$, then clearly the maximum number of correct digits in the results obtained is limited by $p=\min(t,18)$. Because of errors in argument reduction when computing elementary functions inside S17DCF, the actual number of correct digits is limited, in general, by $p-s$, where $s=\max(1,|\log |z||,|\log (\text{nu})|)$ represents the number of digits lost due to the argument reduction. Thus the larger the values of $|z|$ and $\text{nu}$, the less the precision in the result. If S17DCF is called with $N>1$, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DCF with different base values of $\text{(nu)}$ and different $N$, the computed values may not agree exactly. Empirical tests with modest values of $\text{(nu)}$ and $z$ have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments
The time taken by the routine for a call of S17DCF is approximately proportional to the value of N, plus a constant. In general it is much cheaper to call S17DCF with N greater than 1, rather than to make N separate calls to S17DCF.

Paradoxically, for some values of z and (ν), it is cheaper to call S17DCF with a larger value of N than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different N, and the costs in each region may differ greatly.

Note that if the function required is Y(x) or Y'(x), i.e., (ν) = 0.0 or 1.0, where x is real and positive, and only a single unscaled function value is required, then it may be much cheaper to call S17ACF or S17ADF respectively.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order Fu, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine with N = 2 to evaluate the function for orders Fu and Fu + 1, and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DEF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
INTEGER N, NZ, IFAIL
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N)
CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the Bessel function $J_{\nu}(z)$, where $z$ is complex, $-(\pi) < \arg z \leq (\pi)$, and $\nu$ is the real, non-negative order. The $N$-member sequence is generated for orders $\nu$, $(\nu)+1,\ldots,(\nu)+N-1$. Optionally, the sequence is scaled by the factor $e^{-|\text{Im } z|}$.

Note: although the routine may not be called with $\nu$ less than zero, for negative orders the formula $J_{\nu}(z)=J_{\nu}(-iz)\cos((\pi)\nu)-Y_{\nu}(z)\sin((\pi)\nu)$ may be used (for the Bessel function $Y_{\nu}(z)$, see S17DCF).

The routine is derived from the routine CBESJ in Amos [2]. It is based on the relations $J_{\nu}(z)=e^{-(\nu)(\pi)i/2}I_{\nu}(-iz)$, $\Im z \geq 0.0$ and $J_{\nu}(z)=e^{-(\nu)(\pi)i/2}I_{\nu}(iz)$, $\Im z < 0.0$.

The Bessel function $I_{\nu}(z)$ is computed using a variety of techniques depending on the region under consideration.

When $N$ is greater than 1, extra values of $J_{\nu}(z)$ are computed using recurrence relations.

For very large $|z|$ or $(\nu)+N-1)$, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller $|z|$ or $(\nu)+N-1)$, the computation is performed but results are accurate to less than half of machine precision. If $\Im z$ is large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.
4. References


5. Parameters

1: FNU -- DOUBLE PRECISION Input
   On entry: the order, (nu), of the first member of the sequence of functions. Constraint: FNU >= 0.0.

2: Z -- COMPLEX(KIND(1.0D0)) Input
   On entry: the argument z of the functions.

3: N -- INTEGER Input
   On entry: the number, N, of members required in the sequence J (z), J (z), ..., J (z). Constraint: N >= 1.
   \( (\nu) \quad (\nu+1) \quad (\nu+N-1) \)

4: SCALE -- CHARACTER*1 Input
   On entry: the scaling option.
   - If SCALE = 'U', the results are returned unscaled.
   - If SCALE = 'S', the results are returned scaled by the factor \( e^{-|\text{Im}z|} \).
   Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D0)) array Output
   On exit: the N required function values: CY(i) contains J (z), for i=1,2,...,N.
   \( (\nu) + i - 1 \)

6: NZ -- INTEGER Output
   On exit: the number of components of CY that are set to zero due to underflow. If NZ > 0, then elements CY(N-NZ+1), CY(N-NZ+2), ..., CY(N) are set to zero.

7: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry FNU < 0.0,
   or N < 1,
   or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of
overflow, because Im Z is larger than a machine-dependent
threshold value (given in the Users' Note for your
implementation). This error exit can only occur when SCALE =
'U'.

IFAIL= 3
The computation has been performed, but the errors due to
argument reduction in elementary functions make it likely
that the results returned by S17DEF are accurate to less
than half of machine precision. This error exit may occur if
either ABS(Z) or FNU + N - 1 is greater than a machine-
dependent threshold value (given in the Users' Note for
your implementation).

IFAIL= 4
No computation has been performed because the errors due to
argument reduction in elementary functions mean that all
precision in results returned by S17DEF would be lost. This
error exit may occur when either ABS(Z) or FNU + N - 1 is
greater than a machine-dependent threshold value (given in
the Users' Note for your implementation).

IFAIL= 5
No results are returned because the algorithm termination
condition has not been met. This may occur because the
parameters supplied to S17DEF would have caused overflow or
underflow.

7. Accuracy

All constants in subroutine S17DEF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is limited by \( p = \min(t, 18) \). Because of errors in argument reduction when computing elementary functions inside S17DEF, the actual number of correct digits is limited, in general, by \( p - s \), where 
\[
s = \max(1, \log_{10} |z|, \log_{10} (\nu))
\]
represents the number of digits lost due to the argument reduction. Thus the larger the values of \( |z| \) and \( \nu \), the less the precision in the result. If S17DEF is called with \( N > 1 \), then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DEF with different base values of \( \nu \) and different \( N \), the computed values may not agree exactly. Empirical tests with modest values of \( \nu \) and \( z \) have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S17DEF is approximately proportional to the value of \( N \), plus a constant. In general it is much cheaper to call S17DEF with \( N \) greater than 1, rather than to make \( N \) separate calls to S17DEF.

Paradoxically, for some values of \( z \) and \( \nu \), it is cheaper to call S17DEF with a larger value of \( N \) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different \( N \), and the costs in each region may differ greatly.

Note that if the function required is \( J_0(x) \) or \( J_1(x) \), i.e., \( \nu = 0.0 \) or 1.0, where \( x \) is real and positive, and only a single unscaled function value is required, then it may be much cheaper to call S17AEF or S17AFF respectively.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order \( \nu \), the second is a complex value for the argument, \( z \), and the third is a value for the parameter SCALE.

The program calls the routine with \( N = 2 \) to evaluate the function for orders \( \nu \) and \( \nu + 1 \), and it prints the results. The process is repeated until the end of the input data stream is encountered.
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S17 -- Approximations of Special Functions
S17DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
S17DGF returns the value of the Airy function Ai(z) or its derivative Ai'(z) for complex z, with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DGF (DERIV, Z, SCALE, AI, NZ, IFAIL)
INTEGER NZ, IFAIL
COMPLEX(KIND(1.0D0)) Z, AI
CHARACTER*1 DERIV, SCALE
```

3. Description

This subroutine returns a value for the Airy function Ai(z) or its derivative Ai'(z), where z is complex, -(\pi) < arg z <= (\pi).

\[ \text{Ai}(z) = \frac{2z^{\nu/3}}{z^\nu K(\nu)} \]

Optionally, the value is scaled by the factor \( e^{\pi z^{2/3}} \).

The routine is derived from the routine CAIRY in Amos [2]. It is based on the relations

\[ \text{Ai}(z) = \text{------}, \quad \text{Ai}'(z) = \text{---}, \]

\[ (\pi)^{\nu/3}, \quad (\pi)^{\nu/3} \]

where \( K(\nu) \) is the modified Bessel function and \( w=2z^{\nu/3} \).
For very large $|z|$, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller $|z|$, the computation is performed but results are accurate to less than half of machine precision. If $\Re w$ is too large, and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: DERIV -- CHARACTER*1  
   On entry: specifies whether the function or its derivative is required.  
   If DERIV = 'F', $\textrm{Ai}(z)$ is returned.  
   If DERIV = 'D', $\textrm{Ai}'(z)$ is returned.  
   Constraint: DERIV = 'F' or 'D'.

2: Z -- COMPLEX(KIND(1.0D0))  
   On entry: the argument $z$ of the function.

3: SCALE -- CHARACTER*1  
   On entry: the scaling option.  
   If SCALE = 'U', the result is returned unscaled.  
   If SCALE = 'S', the result is returned scaled by the factor $2z/z/3e$.  
   Constraint: SCALE = 'U' or 'S'.

4: AI -- COMPLEX(KIND(1.0D0))  
   On exit: the required function or derivative value.

5: NZ -- INTEGER  
   On exit: NZ indicates whether or not AI is set to zero due to underflow. This can only occur when SCALE = 'U'.  
   If NZ = 0, AI is not set to zero.
If NZ = 1, AI is set to zero.

6: IFAIL -- INTEGER  Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry DERIV /= 'F' or 'D'.

or SCALE /= 'U' or 'S'.

IFAIL = 2
No computation has been performed due to the likelihood of

overflow, because Re w is too large, where w=2\sqrt{Z}/Z/3 -- how
large depends on Z and the overflow threshold of the
machine. This error exit can only occur when SCALE = 'U'.

IFAIL = 3
The computation has been performed, but the errors due to
argument reduction in elementary functions make it likely
that the result returned by S17DGF is accurate to less than
half of machine precision. This error exit may occur if ABS (Z) is greater than a machine-dependent threshold value
(given in the Users' Note for your implementation).

IFAIL = 4
No computation has been performed because the errors due to
argument reduction in elementary functions mean that all
precision in the result returned by S17DGF would be lost.
This error exit may occur if ABS(Z) is greater than a
machine-dependent threshold value (given in the Users' Note
for your implementation).

IFAIL = 5
No result is returned because the algorithm termination
condition has not been met. This may occur because the
parameters supplied to S17DGF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DGF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used $t$, then clearly the maximum number of correct digits in the results obtained is limited by $p = \min(t,18)$. Because of errors in argument reduction when computing elementary functions inside S17DGF, the actual number of correct digits is limited, in general, by $p - s$, where $s = \max(1,|\log |z||)$ represents the number of digits lost due to the argument reduction. Thus the larger the value of $|z|$, the less the precision in the result.

Empirical tests with modest values of $z$, checking relations between Airy functions $\text{Ai}(z)$, $\text{Ai}'(z)$, $\text{Bi}(z)$ and $\text{Bi}'(z)$, have shown errors limited to the least significant 3-4 digits of precision.

8. Further Comments

Note that if the function is required to operate on a real argument only, then it may be much cheaper to call S17AGF or S17AJF.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the parameter DERIV, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine and prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DHF returns the value of the Airy function Bi(z) or its derivative Bi'(z) for complex z, with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DHF (DERIV, Z, SCALE, BI, IFAIL)
INTEGER IFAIL
COMPLEX(KIND(1.0D0)) Z, BI
CHARACTER*1 DERIV, SCALE
```

3. Description

This subroutine returns a value for the Airy function Bi(z) or its derivative Bi'(z), where z is complex, -(pi) < argz <= (pi).

\[ |\text{Re} \left(2z \sqrt{z/3}\right)| \]

Optionally, the value is scaled by the factor \( e^{-\frac{1}{3} - \frac{1}{3} + \frac{1}{3}} \). The routine is derived from the routine CBIRY in Amos [2]. It is based on the relations

\[ Bi(z) = \frac{1}{\sqrt{3}} (I_{-\frac{1}{3}}(w) + I_{\frac{1}{3}}(w)), \text{ and} \]

\[ Bi'(z) = \frac{1}{\sqrt{3}} (I_{-\frac{2}{3}}(w) + I_{\frac{2}{3}}(w)), \]

where \( I \) is the modified Bessel function and \( w = 2z \sqrt{z/3} \). For very large \(|z|\), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \(|z|\), the computation is performed but results are accurate to less than half of machine precision. If \( \text{Re} \ z \) is too large, and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References
5. Parameters

1: DERIV -- CHARACTER*1
   On entry: specifies whether the function or its derivative is required.
   If DERIV = 'F', Bi(z) is returned.
   If DERIV = 'D', Bi'(z) is returned.
   Constraint: DERIV = 'F' or 'D'.

2: Z -- COMPLEX(KIND(1.0D0))
   On entry: the argument z of the function.

3: SCALE -- CHARACTER*1
   On entry: the scaling option.
   If SCALE = 'U', the result is returned unscaled.
   If SCALE = 'S', the result is returned scaled by the factor $e^{-\mid Re(2z\sqrt{z/3})\mid}$.
   Constraint: SCALE = 'U' or 'S'.

4: BI -- COMPLEX(KIND(1.0D0))
   On exit: the required function or derivative value.

5: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry DERIV /= 'F' or 'D'.


or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of overflow, because real(Z) is too large - how large depends on the overflow threshold of the machine. This error exit can only occur when SCALE = 'U'.

IFAIL= 3
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the result returned by S17DHF is accurate to less than half of machine precision. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in the result returned by S17DHF would be lost. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No result is returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DHF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DHF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DHF, the actual number of correct digits is limited, in general, by p-s, where s=max(1,|log |z||) represents the number of digits lost due to the argument reduction. Thus the larger the value of |z|, the less the precision in the result.

Empirical tests with modest values of z, checking relations between Airy functions Ai(z), Ai'(z), Bi(z) and Bi'(z), have shown errors limited to the least significant 3-4 digits of precision.
8. Further Comments

Note that if the function is required to operate on a real argument only, then it may be much cheaper to call S17AHF or S17AKF.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the parameter DERIV, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine and prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
(1) \( f(z) \) or \( f(z) \), where \( z \) is complex, \(-\pi < \text{arg} \ z \leq \pi\), and \( \nu \) is the real, non-negative order. The \( N \)-member sequence is generated for orders \( \nu \), \( \nu+1 \), \ldots, \( \nu+N-1 \).

Optionally, the sequence is scaled by the factor \( e^{\pi i \nu} \) if the \( f(z) \) function is \( f(z) \) or by the factor \( e^{-\pi i \nu} \) if the function is \( f(z) \). 

\( f(z) \).

Note: although the routine may not be called with \( \nu \) less than zero, for negative orders the formulae

\[
(1) \quad (\nu \pi i \nu \pi)(1) \quad (2) \quad -(\nu \pi i \nu \pi)(2)
\]

\[
H(z) = e^{\pi i \nu} H(z), \quad H(z) = e^{-\pi i \nu} H(z)
\]

may be used.

The routine is derived from the routine \textsc{cbesh} in \textsc{amos} [2]. It is based on the relation

\[
(\nu) \quad -p(\nu) \quad -p(\nu)
\]

\[
H(z) = -e^{\nu \pi i} K(z e^{\nu \pi i}),
\]

\[
H(z) = -e^{-\nu \pi i} K(z e^{-\nu \pi i})
\]

where \( p=i \) if \( m=1 \) and \( p=-i \) if \( m=2 \), and the Bessel function \( K(z) \) is computed in the right half-plane only.

Continuation of \( K(z) \) to the left half-plane is computed in terms of the Bessel function \( I(z) \). These functions are evaluated using a variety of different techniques, depending on the region under consideration.

When \( N \) is greater than 1, extra values of \( H(z) \) are computed using recurrence relations.

For very large \(|z|\) or \((\nu+N-1)\), argument reduction will cause total loss of accuracy, and no computation is performed. For slightly smaller \(|z|\) or \((\nu+N-1)\), the computation is performed but results are accurate to less than half of machine precision. If \(|z|\) is very small, near the machine underflow threshold, or
((nu)+N-1) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the kind of functions required.
   (1)
   If M = 1, the functions are $H_{(nu)}(z)$.
   (nu)
   (2)
   If M = 2, the functions are $H_{(nu)}(z)$.
   (nu)
   Constraint: M = 1 or 2.

2: FNU -- DOUBLE PRECISION Input
   On entry: the order, (nu), of the first member of the sequence of functions. Constraint: FNU >= 0.0.

3: Z -- COMPLEX(KIND(1.0D0)) Input
   On entry: the argument z of the functions. Constraint: Z /= (0.0, 0.0).

4: N -- INTEGER Input
   On entry: the number, N, of members required in the sequence
   $(M) (M) (M)$
   $H_{(nu)}$, $H_{(nu)+1}$, ..., $H_{(nu)+N-1}$. Constraint: N >= 1.

5: SCALE -- CHARACTER*1 Input
   On entry: the scaling option.
   If SCALE = 'U', the results are returned unscaled.
   If SCALE = 'S', the results are returned scaled by the factor $e^{-iz}$ when M = 1, or by the factor $e^{iz}$ when M = 2.
   Constraint: SCALE = 'U' or 'S'.
6: CY(N) -- COMPLEX(KIND(1.0D)) array  Output
On exit: the N required function values: CY(i) contains
(M)
H
(nu)+i-1

7: NZ -- INTEGER  Output
On exit: the number of components of CY that are set to zero
due to underflow. If NZ > 0, then if Imz>0.0 and M = 1, or
Imz<0.0 and M = 2, elements CY(1), CY(2),...,CY(NZ) are set
to zero. In the complementary half-planes, NZ simply states
the number of underflows, and not which elements they are.

8: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry M /= 1 and M /= 2,
or  FNU < 0.0,
or  Z = (0.0, 0.0),
or  N < 1,
or  SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of
overflow, because ABS(Z) is less than a machine-dependent
threshold value (given in the Users' Note for your
implementation).

IFAIL= 3
No computation has been performed due to the likelihood of
overflow, because FNU + N - 1 is too large - how large
depends on Z and the overflow threshold of the machine.
CHAPTER 15. CHAPTER N

IFAIL = 4
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S17DLF are accurate to less than half of machine precision. This error exit may occur if either \(\text{ABS}(Z)\) or \(\text{FNU} + N - 1\) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL = 5
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S17DLF would be lost. This error exit may occur when either of \(\text{ABS}(Z)\) or \(\text{FNU} + N - 1\) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL = 6
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DLF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DLF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used \(t\), then clearly the maximum number of correct digits in the results obtained is limited by \(p = \min(t, 18)\). Because of errors in argument reduction when computing elementary functions inside S17DLF, the actual number of correct digits is limited, in general, by \(p - s\), where \(s = \max(1, |\log|z||, |\log(\text{nu})|)\) represents the number of digits lost due to the argument reduction. Thus the larger the values of \(|z|\) and \((\text{nu})\), the less the precision in the result. If S17DLF is called with \(N > 1\), then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DLF with different base values of \((\text{nu})\) and different \(N\), the computed values may not agree exactly. Empirical tests with modest values of \((\text{nu})\) and \(z\) have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S17DLF is approximately proportional to the value of \(N\), plus a constant. In
general it is much cheaper to call S17DLF with \( N \) greater than 1, rather than to make \( N \) separate calls to S17DLF.

Paradoxically, for some values of \( z \) and \( \nu \), it is cheaper to call S17DLF with a larger value of \( N \) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different \( N \), and the costs in each region may differ greatly.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the kind of function, \( M \), the second is a value for the order \( FNU \), the third is a complex value for the argument, \( Z \), and the fourth is a value for the parameter \( SCALE \). The program calls the routine with \( N = 2 \) to evaluate the function for orders \( FNU \) and \( FNU + 1 \), and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
This routine evaluates an approximation to the modified Bessel Function of the second kind $K_0(x)$.

Note: $K_0(x)$ is undefined for $x \leq 0$ and the routine will fail for such arguments.

The routine is based on five Chebyshev expansions:

For $0 < x \leq 1$,

$$K_0(x) = -\ln x + a T_0(t) + b T_1(t), \text{ where } t = 2x - 1;$$

For $1 < x \leq 2$,

$$K_0(x) = e^{-x} c T_0(t), \text{ where } t = 2x - 3;$$

For $2 < x \leq 4$,

$$K_0(x) = e^{-x} d T_0(t), \text{ where } t = x - 3;$$

For $x > 4$,

$$K_0(x) = e^{-x} e^{-9+x} T_0(t), \text{ where } t = \frac{1}{x} - 1;$$

For $x$ near zero, $K_0(x) \approx -(\gamma) - \ln(x^2)$, where $(\gamma)$ denotes Euler’s constant. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.

For large $x$, where there is a danger of underflow due to the smallness of $K_0$, the result is set exactly to zero.

4. References
5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X <= 0.0, K is undefined. On soft failure the routine
0
returns zero.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

\[ \frac{\left| \frac{xK(x)}{K(x)} \right|}{1} \approx \frac{\epsilon}{\delta}. \]

Figure 1 shows the behaviour of the error amplification factor

\[ \left| \frac{xK(x)}{K(x)} \right| \approx \frac{\epsilon}{\delta}. \]
However, if \((\delta)\) is of the same order as machine precision, then rounding errors could make \((\epsilon)\) slightly larger than the above relation predicts.

\[
\left| \frac{1}{\ln x} \right|
\]

For small \(x\), the amplification factor is approximately \(\left| \frac{1}{\ln x} \right|\), which implies strong attenuation of the error, but in general \((\epsilon)\) can never be less than the machine precision.

For large \(x\), \((\epsilon) = x(\delta)\) and we have strong amplification of the relative error. Eventually \(K\), which is asymptotically

\[
\frac{-x}{e}
\]

given by \(\frac{-x}{e}\), becomes so small that it cannot be calculated

\(\sqrt{x}\) without underflow and hence the routine will return zero. Note that for large \(x\) the errors will be dominated by those of the Fortran intrinsic function \(\exp\).

8. Further Comments

For details of the time taken by the routine see the appropriate the Users' Note for your implementation.

9. Example

The example program reads values of the argument \(x\) from a file, evaluates the function at each value of \(x\) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S18 -- Approximations of Special Functions

S18ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

S18ADF returns the value of the modified Bessel Function $K(x)$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S18ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the modified Bessel Function of the second kind $K(x)$.

Note: $K(x)$ is undefined for $x \leq 0$ and the routine will fail for such arguments.

The routine is based on five Chebyshev expansions:

For $0 < x \leq 1$,

$$
K(x) = -x \ln x \sum_{r=0}^{1} \left( a_r \sum_{r=0}^{1} b_r T_{r}(t) \right), \quad t = 2x - 1
$$

For $1 < x \leq 2$,

$$
K(x) = e^{-x} \sum_{r=0}^{2} c_r T_{r}(t), \quad t = 2x - 3
$$

For $2 < x \leq 4$,

$$
K(x) = e^{-x} \sum_{r=0}^{2} d_r T_{r}(t), \quad t = x - 3
$$

For $x > 4$,

$$
K(x) = e^{-x} \sum_{r=0}^{2} e_r T_{r}(t), \quad t = \frac{9-x}{2}
$$
For $x$ near zero, $K(x) \approx -1$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision. For very small $x$ on some machines, it is impossible to calculate without overflow and the routine must fail.

For large $x$, where there is a danger of underflow due to the smallness of $K$, the result is set exactly to zero.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   $X \leq 0.0$, $K$ is undefined. On soft failure the routine returns zero.

IFAIL= 2
   $X$ is too small, there is a danger of overflow. On soft failure the routine returns approximately the largest representable value.

7. Accuracy
Let \( \delta \) and \( \epsilon \) be the relative errors in the argument and result respectively.

If \( \delta \) is somewhat larger than the machine precision (i.e., if \( \delta \) is due to data errors etc), then \( \epsilon \) and \( \delta \) are approximately related by:

\[
\frac{\left| x K (x) - K (x) \right|}{K (x)} \approx \frac{\epsilon}{\delta}.
\]

Figure 1 shows the behaviour of the error amplification factor

\[
\frac{\left| x K (x) - K (x) \right|}{K (x)} \approx \frac{\epsilon}{\delta}.
\]

However if \( \delta \) is of the same order as the machine precision, then rounding errors could make \( \epsilon \) slightly larger than the above relation predicts.

For small \( x \), \( \epsilon \approx \delta \) and there is no amplification of errors.

For large \( x \), \( \epsilon \approx x \delta \) and we have strong amplification of the relative error. Eventually \( K \), which is asymptotically \( 1 - \frac{1}{x} \) given by \( e^{-x} \) \( \sqrt{x} \), becomes so small that it cannot be calculated without underflow and hence the routine will return zero. Note that for large \( x \) the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.
9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

```
S18 -- Approximations of Special Functions
S18AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18AEF returns the value of the modified Bessel Function $I_0(x)$ via the routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S18AEF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the modified Bessel Function of the first kind $I_0(x)$.

Note: $I_0(-x)=I_0(x)$, so the approximation need only consider $x\geq 0$.

The routine is based on three Chebyshev expansions:

For $0<x\leq 4$,

```
x ->',
I_0(x)=e^x > a T_r(t), \text{ where } t=2(-)^{-1};
0 -- r r
r=0
```

For $4<x\leq 12$, 

---

---
\[ I(x) = \begin{cases} e^{-x} & \text{for } x < 8, \\ b(t) & \text{where } t = \frac{x}{4}, \\ r = 0 & \end{cases} \]

For \( x > 12, \)

\[ I(x) = \begin{cases} e^{-12} & \text{for } x \approx 1, \\ c(t) & \text{where } t = 2\left(\frac{x}{r}\right) - 1, \\ r = 0 & \end{cases} \]

For small \( x, \) \( I(x) \approx 1. \) This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For large \( x, \) the routine must fail because of the danger of \( x \) overflow in calculating \( e. \)

4. References


5. Parameters

1: \( X -- \text{DOUBLE PRECISION} \) Input
   On entry: the argument \( x \) of the function.

2: \( IFAIL -- \text{INTEGER} \) Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\( IFAIL = 1 \)

\( X \) is too large. On soft failure the routine returns the approximate value of \( I(x) \) at the nearest valid argument.
7. Accuracy

Let \( \delta \) and \( \epsilon \) be the relative errors in the argument and result respectively.

If \( \delta \) is somewhat larger than the machine precision (i.e., if \( \delta \) is due to data errors etc), then \( \epsilon \) and \( \delta \) are approximately related by:

\[
\frac{|xI(x)|}{|1|} \approx \frac{\epsilon}{\frac{1}{|I(x)|}} (\delta).
\]

Figure 1 shows the behaviour of the error amplification factor

\[
\frac{|xI(x)|}{|1|} \approx \frac{1}{|I(x)|}.
\]

Figure 1
Please see figure in printed Reference Manual

However if \( \delta \) is of the same order as machine precision, then rounding errors could make \( \epsilon \) slightly larger than the above relation predicts.

For small \( x \) the amplification factor is approximately \( \frac{x}{2} \), which implies strong attenuation of the error, but in general \( \epsilon \) can never be less than the machine precision.

For large \( x \), \( \epsilon \approx x \delta \) and we have strong amplification of errors. However the routine must fail for quite moderate values of \( x \), because \( I(x) \) would overflow; hence in practice the loss of accuracy for large \( x \) is not excessive. Note that for large \( x \) the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.
9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S18 -- Approximations of Special Functions

S18AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18AFF returns a value for the modified Bessel Function \( I_1(x) \), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S18AFF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the modified Bessel Function of the first kind \( I_1(x) \).

Note: \( I_{-x} = -I_1(x) \), so the approximation need only consider \( x \geq 0 \).

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 4 \),

\[
I_1(x) =\begin{cases}
\frac{(x)^2}{2(2-x)^2}, & \text{for } 0 < x < 4, \\
\sum_{r=0}^{\infty} a_r T_r(t), & \text{for } 4 < x \leq 12,
\end{cases}
\]

where \( T_r \) is the \( r \)-th Chebyshev polynomial of the first kind, and \( a_r \) and \( t \) are given by

\[
t = 2(2-x)^{-1}, \quad a_r = \frac{(-1)^r}{r!} 2^{2r} \frac{\pi}{2}.
\]
$x \to \infty$

$I(x) = e^{-bT(t)}$, where $t = \frac{x}{r^4}$, $r = 0$

For $x > 12$,

$\frac{x}{e^{bT(t)}}$, where $t = 2\left(\frac{x}{r}\right) - 1$.

For small $x$, $I(x) \approx x$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.

For large $x$, the routine must fail because $I(x)$ cannot be represented without overflow.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument $x$ of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   $X$ is too large. On soft failure the routine returns the approximate value of $I(x)$ at the nearest valid argument.
7. Accuracy

Let \( \delta \) and \( \epsilon \) be the relative errors in the argument and result respectively.

If \( \delta \) is somewhat larger than the machine precision (i.e., if \( \delta \) is due to data errors etc), then \( \epsilon \) and \( \delta \) are approximately related by:

\[
\frac{|xI(x) - I(x)|}{I(x)} \approx \frac{\epsilon}{\delta}.
\]

Figure 1 shows the behaviour of the error amplification factor

\[
\frac{|xI(x) - I(x)|}{I(x)} \approx \frac{\epsilon}{\delta},
\]

Please see figure in printed Reference Manual

However if \( \delta \) is of the same order as machine precision, then rounding errors could make \( \epsilon \) slightly larger than the above relation predicts.

For small \( x \), \( \epsilon \approx \delta \) and there is no amplification of errors.

For large \( x \), \( \epsilon \approx x \delta \) and we have strong amplification of errors. However the routine must fail for quite moderate values of \( x \) because \( I(x) \) would overflow; hence in practice the loss of accuracy for large \( x \) is not excessive. Note that for large \( x \), the errors will be dominated by those of the Fortran intrinsic function \( \text{EXP} \).

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S18 -- Approximations of Special Functions

S18DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18DCF returns a sequence of values for the modified Bessel functions $K_n(z)$ for complex $z$, non-negative $(\nu)$ and $n=0,1,\ldots,N-1$, with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S18DCF (FNU, Z, N, SCALE, CY, NZ, IFAIL)

INTEGER N, NZ, IFAIL
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N)
CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the modified Bessel function $K_\nu(z)$, where $z$ is complex, $-(\pi) < \arg z \leq (\nu)(\pi)$, and $(\nu)$ is the real, non-negative order. The $N$-member sequence is generated for orders $(\nu), (\nu)+1,\ldots,(\nu)+N-1$. Optionally, the sequence is scaled by the factor $e^z$.

The routine is derived from the routine CBESK in Amos [2].

Note: although the routine may not be called with $(\nu)$ less than zero, for negative orders the formula $K_{(-\nu)}(z)$ may be used.

When $N$ is greater than 1, extra values of $K_\nu(z)$ are computed using recurrence relations.
For very large $|z|$ or $(\nu+N-1)$, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller $|z|$ or $(\nu+N-1)$, the computation is performed but results are accurate to less than half of machine precision. If $|z|$ is very small, near the machine underflow threshold, or $(\nu+N-1)$ is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: FNU -- DOUBLE PRECISION Input
   On entry: the order, $(\nu)$, of the first member of the sequence of functions. Constraint: FNU $\geq 0.0$.

2: Z -- COMPLEX(KIND(1.0D0)) Input
   On entry: the argument $z$ of the functions. Constraint: $Z \neq (0.0, 0.0)$.

3: N -- INTEGER Input
   On entry: the number, $N$, of members required in the sequence $K_{(\nu)}(z), K_{(\nu)+1}(z), \ldots, K_{(\nu)+N-1}(z)$. Constraint: $N \geq 1$.

4: SCALE -- CHARACTER*1 Input
   On entry: the scaling option.
   
   If SCALE = 'U', the results are returned unscaled.

   If SCALE = 'S', the results are returned scaled by the $z$ factor $e^z$. Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D)) array Output
   On exit: the $N$ required function values: $CY(i)$ contains $K_{(\nu)+i-1}(z)$, for $i=1,2,\ldots,N$.

6: NZ -- INTEGER Output
   On exit: the number of components of CY that are set to zero.
due to underflow. If NZ > 0 and Rez>=0.0, elements CY(1), CY (2), ..., CY(NZ) are set to zero. If Rez<0.0, NZ simply states the number of underflows, and not which elements they are.

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry FNU < 0.0,
or Z = (0.0, 0.0),
or N < 1,
or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of overflow, because ABS(Z) is less than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 3
No computation has been performed due to the likelihood of overflow, because FNU + N - 1 is too large - how large depends on Z and the overflow threshold of the machine.

IFAIL= 4
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S18DCF are accurate to less than half of machine precision. This error exit may occur if either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S18DCF would be lost. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL = 6
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S18DCF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S18DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by \( p = \min(t, 18) \). Because of errors in argument reduction when computing elementary functions inside S18DCF, the actual number of correct digits is limited, in general, by \( p - s \), where \( s = \max(1, \log |z|, \log |\text{nu}|) \) represents the number of digits lost due to the argument reduction. Thus the larger the values of \(|z|\) and \(|\text{nu}|\), the less the precision in the result. If S18DCF is called with \( N > 1 \), then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S18DCF with different base values of \(|\text{nu}|\) and different \( N \), the computed values may not agree exactly. Empirical tests with modest values of \(|\text{nu}|\) and \( z \) have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S18DCF is approximately proportional to the value of \( N \), plus a constant. In general it is much cheaper to call S18DCF with \( N \) greater than 1, rather than to make \( N \) separate calls to S18DCF.

Paradoxically, for some values of \( z \) and \(|\text{nu}|\), it is cheaper to call S18DCF with a larger value of \( N \) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for
different \( n \), and the costs in each region may differ greatly.

Note that if the function required is \( K(x) \) or \( K_1(x) \), i.e.,
\[
K(n) = 0.0 \text{ or } 1.0, \quad \text{where } x \text{ is real and positive, and only a single function value is required, then it may be much cheaper to call S18ACF, S18ADF, S18CCF(*) or S18CDF(*), depending on whether a scaled result is required or not.}
\]

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order \( FNU \), the second is a complex value for the argument, \( Z \), and the third is a value for the parameter \( SCALE \). The program calls the routine with \( N = 2 \) to evaluate the function for orders \( FNU \) and \( FNU + 1 \), and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S18 -- Approximations of Special Functions
S18DEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18DEF returns a sequence of values for the modified Bessel functions \( I_n(z) \) for complex \( z \), non-negative \( n \) and \( n+1 \) \( n=0,1,\ldots,N-1 \), with an option for exponential scaling.

2. Specification

\[
\text{SUBROUTINE S18DEF (FNU, Z, N, SCALE, CY, NZ, IFAIL)}
\]
\[
\text{INTEGER N, NZ, IFAIL}
\]
\[
\text{DOUBLE PRECISION FNU}
\]
\[
\text{COMPLEX(KIND(1.0D0)) Z, CY(N)}
\]
\[
\text{CHARACTER*1 SCALE}
\]

3. Description
This subroutine evaluates a sequence of values for the modified
Bessel function $I_\eta(z)$, where $z$ is complex, $-(\pi) < \text{arg} z \leq (\pi)$
($\pi$), and ($\eta$) is the real, non-negative order. The $N$-member
sequence is generated for orders ($\eta$), ($\eta$)+1, ..., ($\eta$)+$N$-1.

Optionally, the sequence is scaled by the factor $e^{-|\text{Re} z|}$.

The routine is derived from the routine CBESI in Amos [2].

Note: although the routine may not be called with ($\eta$) less than
zero, for negative orders the formula

$$2 \frac{I_\eta(z)}{I_\eta(z)} = \frac{\sin((\pi)(\eta))}{((\pi)(\eta))}$$

may be used (for

the Bessel function $K_\eta(z)$, see S18DCF).

When $N$ is greater than 1, extra values of $I_\eta(z)$ are computed
using recurrence relations.

For very large $|z|$ or ($\eta$)+$N$-1, argument reduction will cause
total loss of accuracy, and so no computation is performed. For
slightly smaller $|z|$ or ($\eta$)+$N$-1, the computation is performed
but results are accurate to less than half of machine precision.
If $\text{Re}(z)$ is too large and the unscaled function is required,
there is a risk of overflow and so no computation is performed.
In all the above cases, a warning is given by the routine.

4. References

Functions. Dover Publications.

Functions of a Complex Argument and Nonnegative Order. ACM

5. Parameters

1: FNU -- DOUBLE PRECISION
   On entry: the order, ($\eta$), of the first member of the
   sequence of functions. Constraint: FNU $\geq 0.0$.

2: Z -- COMPLEX(KIND(1.0D0))
   On entry: the argument $z$ of the functions.

3: N -- INTEGER
   Input
On entry: the number, \( N \), of members required in the sequence \( I^{(z)}, I^{(z)}(z), \ldots, I^{(z)}(z) \). Constraint: \( N \geq 1 \).

4: \( SCALE \) -- CHARACTER*1 Input
On entry: the scaling option.
If \( SCALE = 'U' \), the results are returned unscaled.
If \( SCALE = 'S' \), the results are returned scaled by the factor \( e^{-|Rez|} \).
Constraint: \( SCALE = 'U' \) or \( 'S' \).

5: \( CY(N) \) -- COMPLEX(KIND(1.0D)) array Output
On exit: the \( N \) required function values: \( CY(i) \) contains \( I^{(z)}(z) \), for \( i=1,2,\ldots,N \).

6: \( NZ \) -- INTEGER Output
On exit: the number of components of \( CY \) that are set to zero due to underflow.
If \( NZ > 0 \), then elements \( CY(N-NZ+1), CY(N-NZ+2), \ldots, CY(N) \) are set to zero.

7: \( IFAIL \) -- INTEGER Input/Output
On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
If on entry \( IFAIL = 0 \) or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

\( IFAIL = 1 \)
On entry \( FNU < 0.0 \),
or \( N < 1 \),
or \( SCALE /= 'U' \) or \( 'S' \).

\( IFAIL = 2 \)
No computation has been performed due to the likelihood of overflow, because \( \text{real}(Z) \) is greater than a machine-
dependent threshold value (given in the Users’ Note for your implementation). This error exit can only occur when SCALE = ’U’.

IFAIL= 3
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S18DEF are accurate to less than half of machine precision. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users’ Note for your implementation).

IFAIL= 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S18DEF would be lost. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users’ Note for your implementation).

IFAIL= 5
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S18DEF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S18DEF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S18DEF, the actual number of correct digits is limited, in general, by p-s, where s=max(1,|log \( |z| \)|,|log \( \nu \)|) represents the number of digits lost due to the argument reduction. Thus the larger the values of \( |z| \) and \( \nu \), the less the precision in the result. If S18DEF is called with N>1, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S18DEF with different base values of \( \nu \) and different N, the computed values may not agree exactly. Empirical tests with modest values of \( \nu \) and z have shown that the discrepancy is limited to the least significant 3-4 digits of precision.
8. Further Comments

The time taken by the routine for a call of S18DEF is approximately proportional to the value of N, plus a constant. In general it is much cheaper to call S18DEF with N greater than 1, rather than to make N separate calls to S18DEF.

Paradoxically, for some values of z and (nu), it is cheaper to call S18DEF with a larger value of N than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different N, and the costs in each region may differ greatly.

Note that if the function required is I (x) or I (x), i.e.,
\[
I(x) = \begin{cases}
0 & \text{if } \nu = 0.0 \\
1 & \text{if } \nu = 1.0
\end{cases}
\]

where x is real and positive, and only a single function value is required, then it may be much cheaper to call S18AEF, S18AFF, S18CEF(*) or S18CFF(*), depending on whether a scaled result is required or not.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order FNU, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine with N = 2 to evaluate the function for orders FNU and FNU + 1, and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S19AAF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Kelvin function \( \text{ber}(x) \).

Note: \( \text{ber}(-x) = \text{ber}(x) \), so the approximation need only consider \( x \geq 0.0 \).

The routine is based on several Chebyshev expansions:

For \( 0 \leq x \leq 5 \),

\[
\text{ber}(x) = \sum_{r=0}^{\infty} a_r T_r(t) \quad \text{with} \quad t = 2(1 - x) - 1;
\]

For \( x > 5 \),

\[
\text{ber}(x) = \frac{e^{-x/\sqrt{2}}}{\sqrt{2\pi}x} \left[ \frac{(1+ \text{a}(t))\cos(\alpha) - \text{b}(t)\sin(\alpha)}{(1+ \text{c}(t))\sin(\beta) - \text{d}(t)\cos(\beta)} \right],
\]

where \( \alpha = \frac{x}{2} \), \( \beta = \frac{x}{2} + \frac{\pi}{4} \), and \( a(t) \), \( b(t) \), \( c(t) \), and \( d(t) \) are expansions in the variable \( t \).
When $x$ is sufficiently close to zero, the result is set directly to $ber\ 0=1.0$.

For large $x$, there is a danger of the result being totally inaccurate, as the error amplification factor grows in an essentially exponential manner; therefore the routine must fail.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument $x$ of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry ABS(X) is too large for an accurate result to be returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than the relative error is important. Let $E$ be the absolute error in the result and $(\delta)$ be the relative error in the argument. If $(\delta)$ is somewhat larger than the machine precision, then we have:

\[
| x | E' = | --(ber x + bei x) | (\delta) \\
| 1 | 1 \\
| \sqrt{2} |
\]

(provided $E$ is within machine bounds).
For small \( x \) the error amplification is insignificant and thus the absolute error is effectively bounded by the machine precision.

For medium and large \( x \), the error behaviour is oscillatory and

\[
\frac{x}{\sqrt{2}} \quad \frac{x}{\sqrt{2}}
\]

its amplitude grows like \( \frac{\sqrt{x}}{2\pi} \). Therefore it is not possible to calculate the function with any accuracy when

\[
\frac{x}{\sqrt{2}} \quad \frac{x}{2\pi}
\]

\( \frac{x}{\sqrt{2}} \quad \frac{x}{2\pi} \). Note that this value of \( x \) is much smaller than the minimum value of \( x \) for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S19 -- Approximations of Special Functions
S19ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S19ABF returns a value for the Kelvin function bei \( x \) via the routine name.

2. Specification
### Description

This routine evaluates an approximation to the Kelvin function $\text{bei} x$.

Note: $\text{bei}(-x)=\text{bei} x$, so the approximation need only consider $x \geq 0.0$.

The routine is based on several Chebyshev expansions:

For $0 \leq x \leq 5$,

$$
\begin{align*}
\text{bei} x &= \frac{2}{\sqrt{x}} \sum_{r=0}^{\infty} a_r T_r \left( \frac{x}{\sqrt{2}} \right), \quad \text{with} \quad t = 2^{2/3} - 1; \\
&= \frac{4}{\sqrt{x}} \sum_{r=0}^{\infty} \frac{\sin \left( \frac{x \sqrt{2}}{8} \right)}{\sqrt{2} \pi x^{1/4}}
\end{align*}
$$

For $x > 5$,

$$
\begin{align*}
\text{bei} x &= \frac{x}{\sqrt{2 \pi}} \left[ (1 + a(t)) \sin \left( \frac{x \sqrt{2}}{8} \right) - b(t) \cos \left( \frac{x \sqrt{2}}{8} \right) \right] \\
&\quad + \frac{x}{\sqrt{2 \pi}} \left[ (1 + c(t)) \cos \left( \frac{x \sqrt{2}}{8} \right) - d(t) \sin \left( \frac{x \sqrt{2}}{8} \right) \right]
\end{align*}
$$

where $a(t), b(t), c(t), d(t)$ are expansions in the variable $t = \frac{\pi}{x}$.
When \( x \) is sufficiently close to zero, the result is computed as
\[
\frac{2}{\sqrt{\pi x}}
\]
\( \text{bei} \ x = \frac{-2}{\sqrt{\pi x}} \). If this result would underflow, the result returned is
\[
\frac{4}{\sqrt{\pi}} \text{beix} = 0.0.
\]
For large \( x \), there is a danger of the result being totally inaccurate, as the error amplification factor grows in an essentially exponential manner; therefore the routine must fail.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION Input
   On entry: the argument \( x \) of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   On entry ABS(X) is too large for an accurate result to be returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than the relative error is important. Let \( E \) be the absolute error in the function, and \( (\delta) \) be the relative error in the argument. If \( (\delta) \) is somewhat larger than the machine precision, then we have:

\[
\frac{| x |}{\sqrt{\pi} \times (-\text{ber} x + \text{bei} x) (\delta)} \leq \frac{1}{\sqrt{2}} \times \frac{1}{(\delta)^2}
\]
For small $x$ the error amplification is insignificant and thus the absolute error is effectively bounded by the machine precision.

For medium and large $x$, the error behaviour is oscillatory and

$$\frac{x}{\sqrt{2}} \sqrt{\frac{x}{\pi}}$$

its amplitude grows like $\frac{1}{\sqrt{2\pi}}$. Therefore it is impossible to calculate the functions with any accuracy when

$$\frac{x}{\sqrt{2}} > \frac{\pi}{\delta}.$$ 

Note that this value of $x$ is much smaller than the minimum value of $x$ for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
2. Specification

   \texttt{DOUBLE PRECISION FUNCTION S19ACF (X, IFAIL)}
   \texttt{INTEGER IFAIL}
   \texttt{DOUBLE PRECISION X}

3. Description

   This routine evaluates an approximation to the Kelvin function \( \text{k}_0(x) \).

   \textbf{Note:} for \( x < 0 \) the function is undefined and at \( x = 0 \) it is infinite so we need only consider \( x > 0 \).

   The routine is based on several Chebyshev expansions:

   For \( 0 < x \leq 1 \),
   \[
   \text{k}_0(x) = -f(t) \log x + \frac{(\pi)^2}{16} g(t) + y(t)
   \]

   where \( f(t) \), \( g(t) \) and \( y(t) \) are expansions in the variable \( t = 2x - 1 \);

   For \( 1 < x \leq 3 \),
   \[
   \text{k}_0(x) = \exp\left(-\frac{x}{16}\right) q(t)
   \]

   where \( q(t) \) is an expansion in the variable \( t = x - 2 \);

   For \( x > 3 \),
   \[
   \text{k}_0(x) = \sqrt{-2x} \left[ \frac{(1 + c(t)) \cos \left(\frac{\pi}{2x} \right) - d(t) \sin \left(\frac{\pi}{2x} \right)}{1 + c(t)} \right]
   \]

   where \( \beta = \frac{\pi}{8} \), \( c(t) \) and \( d(t) \) are expansions in the variable \( t = \frac{x}{2} - 1 \).
When \( x \) is sufficiently close to zero, the result is computed as

\[
\operatorname{ker} x = -\gamma - \log(\gamma) + (\pi - x)
\]

and when \( x \) is even closer to zero, simply as

\[
\operatorname{ker} x = -\gamma - \log(\gamma).
\]

For large \( x \), \( \operatorname{ker} x \) is asymptotically given by

\[
\frac{\pi}{\sqrt{2x}}
\]

this becomes so small that it cannot be computed without underflow and the routine fails.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION Input
   On entry: the argument \( x \) of the function. Constraint: \( X > 0 \).

2: \( IFAIL \) -- INTEGER Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry \( X \) is too large, the result underflows. On soft failure, the routine returns zero.

IFAIL= 2
On entry $X \leq 0$, the function is undefined. On soft failure the routine returns zero.

7. Accuracy

Let $E$ be the absolute error in the result, $(\epsilon)$ be the relative error in the result and $(\delta)$ be the relative error in the argument. If $(\delta)$ is somewhat larger than the machine precision, then we have:

$$
\begin{align*}
| E^- | & = \frac{| x |}{| \text{ker} x + \text{kei} x |} (\delta), \\
& = \frac{1}{\sqrt{2}} \\
& = \frac{\text{ker} x + \text{kei} x}{x} (\delta), \\
(\epsilon)^- & = \frac{1}{\sqrt{2}} \frac{\text{ker} x}{x} (\delta).
\end{align*}
$$

For very small $x$, the relative error amplification factor is approximately given by $\frac{1}{|\log x|}$, which implies a strong attenuation of relative error. However, $(\epsilon)$ in general cannot be less than the machine precision.

For small $x$, errors are damped by the function and hence are limited by the machine precision.

For medium and large $x$, the error behaviour, like the function itself, is oscillatory, and hence only the absolute accuracy for the function can be maintained. For this range of $x$, the amplitude of the absolute error decays like $\frac{(\pi)x}{x\sqrt{2}}$, which implies a strong attenuation of error. Eventually, $\text{ker} x$, which asymptotically behaves like $\frac{-(x/\sqrt{2})}{x}$, becomes so small that it cannot be calculated without causing underflow, and the routine returns zero. Note that for large $x$ the errors are dominated by those of the Fortran intrinsic function EXP.
8. Further Comments

Underflow may occur for a few values of $x$ close to the zeros of $\ker x$, below the limit which causes a failure with IFAIL = 1.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

8. Further Comments

Underflow may occur for a few values of $x$ close to the zeros of $\ker x$, below the limit which causes a failure with IFAIL = 1.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
where \( f(t) \), \( g(t) \) and \( v(t) \) are expansions in the variable \( t = 2x - 1 \);

For \( 1 < x \leq 3 \),

\[
keix = \exp(-x)u(t) \tag{8}
\]

where \( u(t) \) is an expansion in the variable \( t = x - 2 \);

For \( x > 3 \),

\[
keix = \frac{\pi}{\sqrt{2}} \left[ \frac{1}{x} \right]^{1/2} e^{-x/\sqrt{2}} \left[ (1+\frac{c(t)}{x} \sin(\beta)) - \frac{d(t)}{x} \cos(\beta) \right] \tag{9}
\]

where \( \beta = \frac{x}{8} \), and \( c(t) \) and \( d(t) \) are expansions in the variable \( t = \frac{x}{2} \).

For \( x < 0 \), the function is undefined, and hence the routine fails and returns zero.

When \( x \) is sufficiently close to zero, the result is computed as

\[
keix = -\frac{\pi}{4} \left( x \right)^{1/2} \left( 1 - \gamma \right) - \frac{1}{4} \left( \log(-2) \right) \tag{10}
\]

and when \( x \) is even closer to zero simply as

\[
keix = -\frac{\pi}{4} \tag{11}
\]
For large $x$, $keix$ is asymptotically given by $\sqrt{\frac{-e}{2x}}$ and this becomes so small that it cannot be computed without underflow and the routine fails.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION  
   Input  
   On entry: the argument $x$ of the function. Constraint: $X \geq 0$.

2: $IFAIL$ -- INTEGER  
   Input/Output  
   On entry: $IFAIL$ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.  
   On exit: $IFAIL = 0$ unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

$IFAIL = 1$  
On entry $X$ is too large, the result underflows. On soft failure, the routine returns zero.

$IFAIL = 2$  
On entry $X < 0$, the function is undefined. On soft failure the routine returns zero.

7. Accuracy

Let $E$ be the absolute error in the result, and $(\delta)$ be the relative error in the argument. If $(\delta)$ is somewhat larger than the machine representation error, then we have:

$$|x| E^{-1} = \left| -\text{ker} \ x + \text{ksi} \ x \right| (\delta).$$

$$|1| \begin{vmatrix} 1 & 1 \\ \sqrt{2} & 1 \end{vmatrix}$$

For small $x$, errors are attenuated by the function and hence are limited by the machine precision.
For medium and large \( x \), the error behaviour, like the function itself, is oscillatory and hence only absolute accuracy of the function can be maintained. For this range of \( x \), the amplitude of

\[
\frac{e^{-x/\sqrt{2}}}{\pi x - x/\sqrt{2}}
\]

the absolute error decays like \( \frac{e^{-x/\sqrt{2}}}{\sqrt{2}} \), which implies a strong attenuation of error. Eventually, \( keix \), which is

\[
\frac{e^{-x/\sqrt{2}}}{\pi x - x/\sqrt{2}}
\]

asymptotically given by \( \frac{e^{-x/\sqrt{2}}}{\sqrt{2x}} \), becomes so small that it cannot be calculated without causing underflow and therefore the routine returns zero. Note that for large \( x \), the errors are dominated by those of the Fortran intrinsic function \( \text{EXP} \).

8. Further Comments

Underflow may occur for a few values of \( x \) close to the zeros of \( keix \), below the limit which causes a failure with \( \text{IFAIL} = 1 \).

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
2. Specification

DOUBLE PRECISION FUNCTION S20ACF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Fresnel Integral

\[ S(x) = \frac{x}{\pi^2} \int_0^x \sin\left(\frac{t^2}{2}\right) \, dt. \]

Note: \( S(x) = -S(-x) \), so the approximation need only consider \( x \geq 0.0 \).

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 3 \),

\[ S(x) = x \sum_{r=0}^{3} \frac{a_r t^r}{(x^4)^r}, \quad t = 2^{r} - 1, \]

For \( x > 3 \),

\[ S(x) = \frac{1}{2} \left( \frac{\pi}{2} \right)^2 \left( f(x) \cos(\frac{x^2}{2}) - g(x) \sin(\frac{x^2}{2}) \right), \]

where \( f(x) = \sum_{r=0}^{3} b_r t^r \),

and \( g(x) = \sum_{r=0}^{3} c_r t^r \), with \( t = 2^{r} - 1 \).

For small \( x \), \( S(x) \approx \frac{x^6}{6} \). This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision. For very small \( x \), this approximation would underflow; the result is then set exactly to zero.
For large $x$, $f(x) = \frac{1}{\pi}$ and $g(x) = \frac{1}{\pi}$. Therefore for \( \frac{1}{\pi} \),
\[ \frac{1}{\pi} \]
moderately large $x$, when \( \frac{1}{\pi} \) is negligible compared with \( \frac{1}{\pi} \),
\[ \frac{1}{\pi} \]
the second term in the approximation for $x > 3$ may be dropped. For very large $x$, when \( \frac{1}{\pi} \) becomes negligible, $S(x) = \frac{1}{\pi}$. However \( \frac{1}{\pi} \)
there will be considerable difficulties in calculating \( \frac{1}{\pi} \)
\( \cos(\frac{\pi}{2}x) \) accurately before this final limiting value can be
\[ \frac{1}{\pi} \]
used. Since $\cos(\frac{\pi}{2}x)$ is periodic, its value is essentially \( \frac{1}{2} \)
determined by the fractional part of $x$. If $x = N + \theta$, where $N$
\( \frac{1}{\pi} \)
is an integer and $0 \leq \theta < 1$, then $\cos(\frac{\pi}{2}x)$ depends on \( \frac{1}{2} \)
(\( \frac{\pi}{2} \)) \( \theta \) and on $N$ modulo 4. By exploiting this fact, it is possible to retain significance in the calculation of \( \frac{1}{\pi} \)
\( \cos(\frac{\pi}{2}x) \) either all the way to the very large $x$ limit, or at \( \frac{1}{2} \)
least until the integer part of \( \frac{1}{2} \) is equal to the maximum integer allowed on the machine.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
Input
On entry: the argument $x$ of the function.

2: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

\[ \frac{\pi^2}{2} \left| \frac{\sin\left(\frac{\pi}{2}x\right)}{S(x)} \right| \approx (\epsilon) \approx (\delta). \]

Figure 1 shows the behaviour of the error amplification factor

However if (delta) is of the same order as the machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, \( (\epsilon) \approx 3(\delta) \) and hence there is only moderate amplification of relative error. Of course for very small x where the correct result would underflow and exact zero is returned, relative error-control is lost.

For moderately large values of x,

\[ \frac{\pi^2}{4} \left| \frac{\sin\left(\frac{\pi}{4}x\right)}{S(x)} \right| \approx (\epsilon) \approx (\delta). \]
and the result will be subject to increasingly large amplification of errors. However the above relation breaks down for large values of \(x\) (i.e., when \(\frac{1}{x}\) is of the order of the machine precision in this region the relative error in the result is essentially bounded by \(\frac{\pi}{x}\)). Hence the effects of error amplification are limited and at worst the relative error loss should not exceed half the possible number of significant figures.

8. Further Comments

None.

9. Example

The example program reads values of the argument \(x\) from a file, evaluates the function at each value of \(x\) and prints the results. The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S20 -- Approximations of Special Functions
S20ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S20ADF returns a value for the Fresnel Integral \(C(x)\), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S20ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```
3. Description

This routine evaluates an approximation to the Fresnel Integral

\[ C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt. \]

Note: \( C(x) = -C(-x) \), so the approximation need only consider \( x \geq 0.0 \).

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 3 \),

\[ C(x) = x \sum_{r=0}^{(3)} a_r T_r(t), \]

with \( t = 2^{-1}(x) - 1 \).

For \( x > 3 \),

\[ C(x) = \frac{1}{2} f(x) + \frac{1}{3} g(x) - \frac{1}{2} \sin\left(\frac{\pi x}{2}\right) - \frac{1}{3} \cos\left(\frac{\pi x}{2}\right), \]

where \( f(x) = \sum_{r=0}^{(3)} b_r T_r(t) \),

and \( g(x) = \sum_{r=0}^{(3)} c_r T_r(t) \), with \( t = 2^{-1}(x) - 1 \).

For small \( x \), \( C(x) \approx x \). This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For large \( x \), \( f(x) \approx \frac{1}{\pi} \) and \( g(x) \approx \frac{2}{\pi} \). Therefore for \( \frac{1}{\pi} \approx \frac{2}{\pi} \),

moderately large \( x \), when \( \frac{1}{\pi} \) is negligible compared with \( \frac{1}{\pi x} \).
the second term in the approximation for $x > 3$ may be dropped. For very large $x$, when $\frac{1}{(\pi x)^{2/2}}$ becomes negligible, $C(x)^2 = -$. However there will be considerable difficulties in calculating $\sin(\frac{\pi x}{2})$ accurately before this final limiting value can be used. Since $\sin(\frac{\pi x}{2})$ is periodic, its value is essentially determined by the fractional part of $x$. If $x = N + \theta$, where $N$ is an integer and $0 \leq \theta < 1$, then $\sin(\frac{\pi x}{2})$ depends on $\theta$ and on $N$ modulo 4. By exploiting this fact, it is possible to retain some significance in the calculation of $\sin(\frac{\pi x}{2})$ either all the way to the very large $x$ limit, or at least until the integer part of $-\frac{x}{2}$ is equal to the maximum integer allowed on the machine.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION Input
On entry: the argument $x$ of the function.

2: $IFAIL$ -- INTEGER Input/Output
On entry: $IFAIL$ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: $IFAIL = 0$ unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter $IFAIL$ has been included for consistency with other routines in this chapter.
7. Accuracy

Let \( (\delta) \) and \( (\epsilon) \) be the relative errors in the argument and result respectively.

If \( (\delta) \) is somewhat larger than the machine precision (i.e., if \( (\delta) \) is due to data errors etc), then \( (\epsilon) \) and \( (\delta) \) are approximately related by:

\[
(\epsilon) \approx \left| \frac{(\pi/2) x \cos\left(-\frac{x}{2}\right)}{C(x)} \right| \delta.
\]

Figure 1 shows the behaviour of the error amplification factor

\[
(\epsilon) \approx \left| \frac{(\pi/2) x \cos\left(-\frac{x}{2}\right)}{C(x)} \right| \delta.
\]

However if \( (\delta) \) is of the same order as the machine precision, then rounding errors could make \( (\epsilon) \) slightly larger than the above relation predicts.

For small \( x \), \( (\epsilon) \approx (\delta) \) and there is no amplification of relative error.

For moderately large values of \( x \),

\[
(\epsilon) \approx 2 x \cos\left(-\frac{x}{2}\right) |(\delta)|
\]

and the result will be subject to increasingly large amplification of errors. However the above relation breaks down for large values of \( x \) (i.e., when \( \frac{1}{x} \) is of the order of the machine precision); in this region the relative error in the result is essentially bounded by \( \frac{\pi}{x} \).
Hence the effects of error amplification are limited and at worst the relative error loss should not exceed half the possible number of significant figures.

8. Further Comments

None.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S21 -- Approximations of Special Functions
S21BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S21BAF returns a value of an elementary integral, which occurs as a degenerate case of an elliptic integral of the first kind, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S21BAF (X, Y, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y
```

3. Description

This routine calculates an approximate value for the integral

\[
\int_{C}^{\infty} \frac{1}{\sqrt{t^2 + x(t+y)}} \, dt
\]
where \( x \geq 0 \) and \( y \neq 0 \).

This function, which is related to the logarithm or inverse hyperbolic functions for \( y < x \) and to inverse circular functions if \( x < y \), arises as a degenerate form of the elliptic integral of the first kind. If \( y < 0 \), the result computed is the Cauchy principal value of the integral.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the system:

\[
\begin{align*}
x &= x, \\
y &= y, \\
(\mu)_n &= (x + 2y) / 3, \\
S_n &= (y - x) / 3(\mu), \\
\lambda_n &= y + 2 / x y, \\
x_{n+1} &= (x + (\lambda)_n) / 4, \\
y_{n+1} &= (y + (\lambda)_n) / 4.
\end{align*}
\]

The quantity \( |S| \) for \( n = 0, 1, 2, 3, \ldots \) decreases with increasing \( n \), and eventually \( |S| \approx 1/4 \). For small enough \( S \) the required function value can be approximated by the first few terms of the Taylor series about the mean. That is

\[
R(x,y) = \left( 1 + \frac{2^3}{3 \times S} + \frac{3^3}{3^2 S} + \frac{4^3}{3^3 S^2} + \frac{5^3}{3^4 S^3} \right) / \mu.
\]

The truncation error involved in using this approximation is
bounded by $16|S|/(1-2|S|)$ and the recursive process is stopped when $S$ is small enough for this truncation error to be negligible compared to the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the result is done before returning to the calling program.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input
   On entry: the arguments x and y of the function, respectively. Constraint: X $\geq$ 0.0 and Y $\neq$ 0.0.

3: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).
IFAIL= 1
On entry $X < 0.0$; the function is undefined.

IFAIL= 2
On entry $Y = 0.0$; the function is undefined.

On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be excessive as the algorithm does not involve any significant amplification of round-off error. It is reasonable to assume that the result is accurate to within a small multiple of the machine precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
DOUBLE PRECISION FUNCTION S21BBF (X, Y, Z, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y, Z

3. Description

This routine calculates an approximation to the integral

\[ R(x, y, z) = \int_{0}^{\infty} \frac{1}{(t+x)(t+y)(t+z)} \, dt \]

where \( x, y, z \geq 0 \) and at most one is zero.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:

\[
\begin{align*}
x &= \min(x, y, z), z = \max(x, y, z), \\
y &= \text{remaining third intermediate value argument}.
\end{align*}
\]

(This ordering, which is possible because of the symmetry of the function, is done for technical reasons related to the avoidance of overflow and underflow.)

\[
\begin{align*}
(\mu) &= \frac{(x + y + 3z)}{3} \\
X &= \frac{(1-x)}{(\mu)} \\
Y &= \frac{(1-y)}{(\mu)} \\
Z &= \frac{(1-z)}{(\mu)}
\end{align*}
\]

\[
(\lambda) = \frac{x + y + z + x}{x} \\
x = \frac{(x + (\lambda))}{4}
\]
\[ y_{n+1} = \frac{y_n + \lambda}{4} \]
\[ z_{n+1} = \frac{z_n + \lambda}{4} \]

\((\varepsilon)_{\text{max}} = \max(|X|, |Y|, |Z|)\) and the function may be approximated adequately by a 5th order power series:

\[
R(x, y, z) = \frac{1}{F(\mu)} \left( \frac{E_{12}E_{23}E_{31}}{10 \ 24 \ 44 \ 14} \right) - \frac{1 - E_{12} - E_{23} - E_{31}}{10 \ 24 \ 44 \ 14} \]

where \(E = X \cdot Y + Y \cdot Z + Z \cdot X\), \(E_1 = X \cdot Y \cdot Z\).

The truncation error involved in using this approximation is bounded by \((\varepsilon)_{\text{max}} / 4(1 - (\varepsilon)_{\text{max}})\) and the recursive process is stopped when this truncation error is negligible compared with the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the result is done before returning to the calling program.

4. References


5. Parameters
1: X -- DOUBLE PRECISION  Input
2: Y -- DOUBLE PRECISION  Input
3: Z -- DOUBLE PRECISION  Input
   On entry: the arguments x, y and z of the function.
   Constraint: X, Y, Z >= 0.0 and only one of X, Y and Z may be
               zero.
4: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
             familiar with this parameter (described in the Essential
             Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
            Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry one or more of X, Y and Z is negative; the function
   is undefined.

IFAIL= 2
   On entry two or more of X, Y and Z are zero; the function is
   undefined.

   On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.
If two arguments are equal, the function reduces to the elementary integral \( C \), computed by S21BAF.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S21 -- Approximations of Special Functions
S21BCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S21BCF returns a value of the symmetrised elliptic integral of the second kind, via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S21BCF (X, Y, Z, IFAIL)}
\]
\[
\text{INTEGER IFAIL}
\]
\[
\text{DOUBLE PRECISION X, Y, Z}
\]

3. Description

This routine calculates an approximate value for the integral

\[
\int_{\mathcal{D}} \frac{1}{(t+x)(t+y)(t+z)} \, dt
\]

where \( x, y \geq 0 \), at most one of \( x \) and \( y \) is zero, and \( z > 0 \).

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:
\[
x = x \\
y = y \\
z = z \\
\]

\[
(x + y + 3z) / 5 \\
\]

\[
X = \frac{1-x}{(mu)} \\
Y = \frac{1-y}{(mu)} \\
Z = \frac{1-z}{(mu)} \\
\]

\[
(x + (lambda)) / 4 \\
\]

\[
(y + (lambda)) / 4 \\
\]

\[
(z + (lambda)) / 4 \\
\]

For \( n \) sufficiently large,

\[
\epsilon = \max(|X|, |Y|, |Z|)^{\frac{1}{4}} \\
\]

and the function may be approximated adequately by a 5th order

\[
R(x,y,z) = \sum_{m=0}^{\infty} \frac{z + (lambda)}{z} / n \\
\]
\[ \begin{array}{ccccccccc}
2 & 4 & 3 & 2 & 1 & 3 & 2 & 3 & 4 \\
\hline
1 & -3 & -3 & -3 & -3 & -3 & -3 & -3 & -3 \\
\end{array} \]

\[-n \]

\[ \begin{array}{cccccccc}
7 & n & 3 & n & 22 & n & 11 & n & 13 & n \\
\hline
\end{array} \]

\[ / 3 \]

\[ / (\mu) \]

\[ \sqrt{n} \]

where

\[ (m) ( m m m) \]

\[ S = (X + Y + 3Z) / 2m. \]

\[ n ( n n n) \]

The truncation error in this expansion is bounded by

\[ 6 \]

\[ 3(\epsilon) \]

\[ n \]

---------- and the recursive process is terminated when

\[ / 3 \]

\[ / (1-\epsilon) \]

\[ \sqrt{n} \]

this quantity is negligible compared with the machine precision.

The routine may fail either because it has been called with arguments outside the domain of definition, or with arguments so extreme that there is an unavoidable danger of setting underflow or overflow.

\[-3/2 \]

Note: \( R(x,x,x) = x \), so there exists a region of extreme arguments for which the function value is not representable.

4. References


5. Parameters
1: X -- DOUBLE PRECISION  
   Input

2: Y -- DOUBLE PRECISION  
   Input

3: Z -- DOUBLE PRECISION  
   Input
   On entry: the arguments x, y and z of the function.
   Constraint: X, Y >= 0.0, Z > 0.0 and only one of X and Y may
   be zero.

4: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry either X or Y is negative, or both X and Y are
   zero; the function is undefined.

IFAIL= 2
   On entry Z <= 0.0; the function is undefined.

IFAIL= 3
   On entry either Z is too close to zero or both X and Y are
   too close to zero: there is a danger of setting overflow.

IFAIL= 4
   On entry at least one of X, Y and Z is too large: there is a
   danger of setting underflow.
   On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S21BDF returns a value of the symmetrised elliptic integral of the third kind, via the routine name.

2. Specification

DOUBLE PRECISION FUNCTION S21BDF (X, Y, Z, R, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y, Z, R

3. Description

This routine calculates an approximation to the integral

\[
\int_{\frac{3}{2}}^{\infty} \frac{dt}{\sqrt{J(t+(\rho))/(t+x)(t+y)(t+z)}}
\]

where \(x, y, z\geq 0\), \((\rho)\neq 0\) and at most one of \(x, y\) and \(z\) is zero.
If \( p < 0 \), the result computed is the Cauchy principal value of the integral.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:

\[
\begin{align*}
x &= x_0 \\
y &= y_0 \\
z &= z_0 \\
(\rho) &= (\rho)_0 \\
\mu &= \frac{x + y + z + 2(\rho)}{5} \\
X &= \frac{1-x}{\mu} \\
Y &= \frac{1-y}{\mu} \\
Z &= \frac{1-z}{\mu} \\
P &= \frac{1-(\rho)}{\mu} \\
\lambda &= \frac{x y + y z + z x}{x y + x z + x z} \\
x &= \frac{x + \lambda}{4} \\
y &= \frac{y + \lambda}{4} \\
z &= \frac{z + \lambda}{4} \\
(\rho) &= \frac{(\rho) + \lambda}{4}
\end{align*}
\]
(alpha) = [(rho) (n \ x + y + z )/ n \ n n n n]

(beta) = (rho) ((rho) + (lambda) )

For n sufficiently large,

1
(epsilon) = max(|X|,|Y|,|Z|,|P|)^1 --

and the function may be approximated by a 5th order power series

\[
R(x,y,z,(\rho))=\sum_{m=0}^{n-1} \left[ \frac{3(2)\ 1(3)\ 3(2)\ 2(3)\ 3(2)\ 3(5)}{7n\ 3n\ 22n\ 11n\ 13n\ n}\right]
\]

\[
\frac{\mu}{(m)}\ m\ m\ m\ m\ m\ n
\]

where \( S = (X + Y + Z + 2P) / 2m. \)

The truncation error in this expansion is bounded by

\[
3(\epsilon) / (1 - (\epsilon)) \text{ and the recursion process is terminated when this quantity is negligible compared with the machine precision. The routine may fail either because it has been called with arguments outside the domain of definition or with arguments so extreme that there is an unavoidable danger of setting underflow or overflow.}
\]
Note: \( R(x,x,x,x) = x \), so there exists a region of extreme
arguments for which the function value is not representable.

4. References

Functions. Dover Publications.

Duplication. (Preprint) Department of Physics, Iowa State
University.


5. Parameters

1: X -- DOUBLE PRECISION
Input

2: Y -- DOUBLE PRECISION
Input

3: Z -- DOUBLE PRECISION
Input

4: R -- DOUBLE PRECISION
Input
On entry: the arguments x, y, z and (rho) of the function.
Constraint: X, Y, Z \( \geq 0.0 \), R \( /= 0.0 \) and at most one of X, Y
and Z may be zero.

5: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry at least one of X, Y and Z is negative, or at least
two of them are zero; the function is undefined.
IFAIL = 2
On entry R = 0.0; the function is undefined.

IFAIL = 3
On entry either R is too close to zero, or any two of X, Y and Z are too close to zero; there is a danger of setting overflow.

IFAIL = 4
On entry at least one of X, Y, Z and R is too large; there is a danger of setting underflow.

IFAIL = 5
An error has occurred in a call to S21BAF. Any such occurrence should be reported to NAG.

On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be excessive as the algorithm does not involve any significant amplification of round-off error. It is reasonable to assume that the result is accurate to within a small multiple of the machine precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

If the argument R is equal to any of the other arguments, the function reduces to the integral R, computed by S21BCF.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagSpecialFunctionsPackage (NAGS)

Exports:

s01eaf s13aaf s13acf s13adf s14aaf
s14abf s14baf s15adf s15aef s17acf
s17adff s17aef s17aff s17agf s17ahf
s17ajf s17akf s17dch s17df s17dgf
s17dhf s17dlf s18acf s18adf s18aef
s18aff s18dcf s18dch s19aaf s19abf
s19acf s19adff s1920acff s1920adf s19baf
s21baf s21bchf s21bdf

--- package NAGS NagSpecialFunctionsPackage ---

)abbrev package NAGS NagSpecialFunctionsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:44 1994
++ Description:
++ This package uses the NAG Library to compute some commonly
++ occurring physical and mathematical functions.

NagSpecialFunctionsPackage(): Exports == Implementation where
S == Symbol
FOP ==> FortranOutputStackPackage

Exports == with
s01eaf : (Complex DoubleFloat,Integer) -> Result
++ s01eaf(z,ifail)
++ S01EAF evaluates the exponential function exp(z), for complex z.
++ See \downlink{Manual Page}{manpageXXs01eaf}.
s13aaf : (DoubleFloat,Integer) -> Result
++ s13aaf(x,ifail)
++ returns the value of the exponential integral
++ E (x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs13aaf}.
s13acf : (DoubleFloat,Integer) -> Result
++ s13acf(x,ifail)
++ returns the value of the cosine integral
++ See \downlink{Manual Page}{manpageXXs13acf}.
s13adf : (DoubleFloat,Integer) -> Result
++ s13adf(x,ifail)
++ returns the value of the sine integral
++ See \downlink{Manual Page}{manpageXXs13adf}.
s14aaf : (DoubleFloat,Integer) -> Result
++ s14aaf(x,ifail) returns the value of the Gamma function \(\Gamma(x)\),
++ via the routine name.
++ See \downlink{Manual Page}{manpageXXs14aaf}.
s14abf : (DoubleFloat,Integer) -> Result
++ s14abf(x,ifail) returns a value for the log, \(\ln(\Gamma(x))\), via
++ the routine name.
++ See \downlink{Manual Page}{manpageXXs14abf}.
s14baf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s14baf(a,x,tol,ifail)
++ computes values for the incomplete gamma functions \(P(a,x)\)
++ and \(Q(a,x)\).
++ See \downlink{Manual Page}{manpageXXs14baf}.
s15adf : (DoubleFloat,Integer) -> Result
++ s15adf(x,ifail)
++ returns the value of the complementary error function,
++ \(\text{erfc}(x)\), via the routine name.
++ See \downlink{Manual Page}{manpageXXs15adf}.
s15aef : (DoubleFloat,Integer) -> Result
++ s15aef(x,ifail)
++ returns the value of the error function \(\text{erf}(x)\), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs15aef}.
s17acf : (DoubleFloat,Integer) -> Result
++ s17acf(x,ifail)
++ returns the value of the Bessel Function
++ \(Y(x)\), via the routine name.
++ 0
++ See \downlink{Manual Page}{manpageXXs17acf}.
s17adf : (DoubleFloat,Integer) -> Result
++ s17adf(x,ifail)
++ returns the value of the Bessel Function
++ \(Y(x)\), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs17adf}.
s17aef : (DoubleFloat,Integer) -> Result
++ s17aef(x,ifail)
++ returns the value of the Bessel Function
++ \(J(x)\), via the routine name.
PACKAGE NAGS NAGSPECIALFUNCTIONS PACKAGE

++ 0
++ See \downlink{Manual Page}{manpageXXs17aef}. s17aff : (DoubleFloat, Integer) -> Result ++ s17aff(x, ifail)
++ returns the value of the Bessel Function ++ J(x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs17aff}. s17agf : (DoubleFloat, Integer) -> Result ++ s17agf(x, ifail)
++ returns a value for the Airy function, Ai(x), via the ++ routine name.
++ See \downlink{Manual Page}{manpageXXs17agf}. s17ahf : (DoubleFloat, Integer) -> Result ++ s17ahf(x, ifail)
++ returns a value of the Airy function, Bi(x), via the ++ routine name.
++ See \downlink{Manual Page}{manpageXXs17ahf}. s17ajf : (DoubleFloat, Integer) -> Result ++ s17ajf(x, ifail)
++ returns a value of the derivative of the Airy function ++ Ai(x), via the routine name.
++ See \downlink{Manual Page}{manpageXXs17ajf}. s17akf : (DoubleFloat, Integer) -> Result ++ s17akf(x, ifail)
++ returns a value for the derivative of the Airy function ++ Bi(x), via the routine name.
++ See \downlink{Manual Page}{manpageXXs17akf}. s17dcf : (DoubleFloat, Complex DoubleFloat, Integer, String, _ ++ Integer) -> Result ++ s17dcf(fnu, z, n, scale, ifail)
++ returns a sequence of values for the Bessel functions ++ Y_n(z) for complex z, non-negative (nu) and n=0,1,...,N-1,
++ with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs17dcf}. s17def : (DoubleFloat, Complex DoubleFloat, Integer, String, _ ++ Integer) -> Result ++ s17def(fnu, z, n, scale, ifail)
++ returns a sequence of values for the Bessel functions ++ J_n(z) for complex z, non-negative (nu) and n=0,1,...,N-1,
++ with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs17def}. s17dgf : (String, Complex DoubleFloat, String, Integer) -> Result ++ s17dgf(deriv, z, scale, ifail)
++ returns the value of the Airy function Ai(z) or its ++ derivative Ai'(z) for complex z, with an option for exponential ++ scaling.
++ See \downlink{Manual Page}{manpageXXs17dgf}.
\text{CHAPTER 15. \textsc{CHAPTER N}}

\texttt{s17dhf : (String,Complex DoubleFloat,String,Integer) -> Result}\n\texttt{\quad \text{s17dhf(deriv,z,scale,ifail)}}\n\texttt{\quad \text{returns the value of the Airy function \(Bi(z)\) or its}}\n\texttt{\quad \text{derivative \(Bi'(z)\) for complex \(z\), with an option for exponential}}\n\texttt{\quad \text{scaling.}}\n\texttt{\quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs17dhf}}.}}\n
\texttt{s17dlf : (Integer,DoubleFloat,Complex DoubleFloat,Integer,_}\n\quad \texttt{\quad String,Integer) -> Result}\n\texttt{\quad \text{s17dlf(m,fnu,z,n,\text{scale},ifail)}}\n\texttt{\quad \text{returns a sequence of values for the Hankel functions}}\n\texttt{\quad \begin{align*}\text{(1)} & \quad H^{(1)}_{\nu}(z) \quad \text{or} \quad H^{(2)}_{\nu}(z) \quad \text{for complex} \quad z, \quad \text{non-negative} \quad (\nu) \quad \text{and} \\
\text{(2)} & \quad (\nu)+n \quad (\nu)+n \quad \text{with an option for exponential scaling.}}\n\text{\quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs17dlf}}.}}\n
\texttt{s18acf : (DoubleFloat,Integer) -> Result}\n\texttt{\quad \text{s18acf(x,ifail)}}\n\texttt{\quad \text{returns the value of the modified Bessel Function}}\n\texttt{\quad \begin{align*}\text{0} & \quad K_{0}(x), \quad \text{via the routine name.} \\
\text{1} & \quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs18acf}}.}}\n
\texttt{s18adf : (DoubleFloat,Integer) -> Result}\n\texttt{\quad \text{s18adf(x,ifail)}}\n\texttt{\quad \text{returns the value of the modified Bessel Function}}\n\texttt{\quad \begin{align*}\text{0} & \quad K_{1}(x), \quad \text{via the routine name.} \\
\text{1} & \quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs18adf}}.}}\n
\texttt{s18aef : (DoubleFloat,Integer) -> Result}\n\texttt{\quad \text{s18aef(x,ifail)}}\n\texttt{\quad \text{returns the value of the modified Bessel Function}}\n\texttt{\quad \begin{align*}\text{0} & \quad I_{0}(x), \quad \text{via the routine name.} \\
\text{1} & \quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs18aef}}.}}\n
\texttt{s18aff : (DoubleFloat,Integer) -> Result}\n\texttt{\quad \text{s18aff(x,ifail)}}\n\texttt{\quad \text{returns a value for the modified Bessel Function}}\n\texttt{\quad \begin{align*}\text{0} & \quad I_{1}(x), \quad \text{via the routine name.} \\
\text{1} & \quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs18aff}}.}}\n
\texttt{s18dcf : (DoubleFloat,Complex DoubleFloat,Integer,String,}\n\quad \texttt{\quad String,Integer) -> Result}\n\texttt{\quad \text{s18dcf(fnu,z,n,\text{scale},ifail)}}\n\texttt{\quad \text{returns a sequence of values for the modified Bessel functions}}\n\texttt{\quad \begin{align*}\text{0} & \quad K_{0}(z), \quad \text{for complex} \quad z, \quad \text{non-negative} \quad (\nu) \quad \text{and} \\
\text{1} & \quad (\nu)+n \quad (\nu)+n \quad \text{with an option for exponential scaling.}}\n\text{\quad \text{See \texttt{\textbackslash downlink{Manual Page}{manpageXXs18dcf}}.}}\n
\texttt{s18def : (DoubleFloat,Complex DoubleFloat,Integer,String,}\n\quad \texttt{\quad String,Integer) -> Result}\n\texttt{\quad \text{s18def(fnu,z,n,\text{scale},ifail)}}
++ returns a sequence of values for the modified Bessel functions
++ \( I^{(\nu)}(z) \) for complex \( z \), non-negative \( \nu \) and
++ \( n=0,1,\ldots,N-1 \), with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs18def}.

s19aaf : (DoubleFloat,Integer) -> Result
++ s19aaf(x,ifail)
++ returns a value for the Kelvin function \( \text{ber}(x) \) via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs19aaf}.

s19abf : (DoubleFloat,Integer) -> Result
++ s19abf(x,ifail)
++ returns a value for the Kelvin function \( \text{bei}(x) \) via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs19abf}.

s19acf : (DoubleFloat,Integer) -> Result
++ s19acf(x,ifail)
++ returns a value for the Kelvin function \( \text{ker}(x) \), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs19acf}.

s19adf : (DoubleFloat,Integer) -> Result
++ s19adf(x,ifail)
++ returns a value for the Kelvin function \( \text{kei}(x) \) via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs19adf}.

s20acf : (DoubleFloat,Integer) -> Result
++ s20acf(x,ifail)
++ returns a value for the Fresnel Integral \( S(x) \), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs20acf}.

s20adf : (DoubleFloat,Integer) -> Result
++ s20adf(x,ifail)
++ returns a value for the Fresnel Integral \( C(x) \), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs20adf}.

s21baf : (DoubleFloat,DoubleFloat,Integer) -> Result
++ s21baf(x,y,ifail)
++ returns a value of an elementary integral, which occurs as
++ a degenerate case of an elliptic integral of the first kind, via
++ the routine name.
++ See \downlink{Manual Page}{manpageXXs21baf}.

s21bbf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s21bbf(x,y,z,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the first kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bbf}.

s21bcf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s21bcf(x,y,z,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the second kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bcf}.

\begin{verbatim}
s21bdf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
    Integer) -> Result
++ s21bdf(x,y,z,r,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the third kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bdf}.
\end{verbatim}

Implementation ==

\begin{verbatim}
import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Complex DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(String)

s01eaf(zArg:Complex DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "s01eaf",_
    ["z"::S,"ifail"::S]$Lisp,_
    []$Lisp,_
    [["integer"::S,"ifail"::S]$Lisp_,
    ["double complex"::S,"s01eafResult"::S,"z"::S]$Lisp_]
    )$Lisp_,
    ["s01eafResult"::S,"ifail"::S]$Lisp_,
    [[zArg::Any,ifailArg::Any ]]
    pretend List (Record(key:Symbol,entry:Any)))$Result

s13aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "s13aaf",_
    ["x"::S,"ifail"::S]$Lisp,_
    []$Lisp,_
    ["double"::S,"s13aafResult"::S,"x"::S]$Lisp_,
    ["integer"::S,"ifail"::S]$Lisp_]
    )$Lisp_,
    ["s13aafResult"::S,"ifail"::S]$Lisp_,
    [(xArg::Any,ifailArg::Any )]
    pretend List (Record(key:Symbol,entry:Any)))$Result

s13acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "s13acf",_
    ["x"::S,"ifail"::S]$Lisp,_
    []$Lisp,_
    ["double"::S,"s13acfResult"::S,"x"::S]$Lisp_,
    ["integer"::S,"ifail"::S]$Lisp_]
    )$Lisp_,
    ["s13acfResult"::S,"ifail"::S]$Lisp_,
    [(xArg::Any,ifailArg::Any )]
    pretend List (Record(key:Symbol,entry:Any)))$Result
\end{verbatim}
s13acf(xArg::DoubleFloat, ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"s13acf", _
"x":S,"ifail":S]$Lisp,_
[]$Lisp,_
["double":S,"s13acfResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp,_
["s13acfResult":S,"ifail":S]$Lisp,_
[(xArg::Any,ifailArg::Any )]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

s13adf(xArg::DoubleFloat, ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"s13adf", _
"x":S,"ifail":S]$Lisp,_
[]$Lisp,_
["double":S,"s13adfResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp,_
["s13adfResult":S,"ifail":S]$Lisp,_
[(xArg::Any,ifailArg::Any )]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

s14aaf(xArg::DoubleFloat, ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"s14aaf", _
"x":S,"ifail":S]$Lisp,_
[]$Lisp,_
["double":S,"s14aafResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp,_
["s14aafResult":S,"ifail":S]$Lisp,_
[(xArg::Any,ifailArg::Any )]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

s14abf(xArg::DoubleFloat, ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"s14abf", _
"x":S,"ifail":S]$Lisp,_
[]$Lisp,_
["double":S,"s14abfResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp,_
["s14abfResult":S,"ifail":S]$Lisp,_
[(xArg::Any,ifailArg::Any )]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
s14baf(aArg:DoubleFloat,xArg:DoubleFloat,tolArg:DoubleFloat,_
  ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "s14baf",_,
    "a":S,"x":S,"tol":S,"p":S,"q":S_,
    "ifail":S]$Lisp,_
    "p"::S,"q"::S]$Lisp_,
    "q":S]$Lisp_,
  ,["integer":S,"ifail":S]$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))]]$Result

s15adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "s15adf",_,
    "x":S,"ifail":S]$Lisp_,
  []$Lisp_,
  [["double":S,"s15adfResult":S,"x":S]$Lisp_,
    ,["integer":S,"s15adfResult":S]$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))]]$Result

s15aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "s15aef",_,
    "x":S,"ifail":S]$Lisp_,
  []$Lisp_,
  [["double":S,"s15aefResult":S,"x":S]$Lisp_,
    ,["integer":S,"s15aefResult":S]$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))]]$Result

s17acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
    "s17acf",_,
    "x":S,"ifail":S]$Lisp_,
  []$Lisp_,
  [["double":S,"s17acfResult":S,"x":S]$Lisp_,
    ,["integer":S,"s17acfResult":S]$Lisp_)
  pretend List (Record(key:Symbol,entry:Any))]]$Result
s17adf(xArg:DoubleFloat, ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "s17adf",_
  ["x":S,"ifail":S]$Lisp,_
  []$Lisp,_
  [["double":S,"s17adfResult":S,"x":S]$Lisp_,
    ["integer":S,"ifail":S]$Lisp_]
  $Lisp_,
  ["s17adfResult":S,"ifail":S]$Lisp_,
  [(xArg::Any, ifailArg::Any )]_
  $List Any]$Lisp
  pretend List (Record(key:Symbol, entry:Any))]
$Result

s17aef(xArg:DoubleFloat, ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "s17aef",_
  ["x":S,"ifail":S]$Lisp,_
  []$Lisp,_
  [["double":S,"s17aefResult":S,"x":S]$Lisp_,
    ["integer":S,"ifail":S]$Lisp_]
  $Lisp_,
  ["s17aefResult":S,"ifail":S]$Lisp_,
  [(xArg::Any, ifailArg::Any )]_
  $List Any]$Lisp
  pretend List (Record(key:Symbol, entry:Any))]
$Result

s17aff(xArg:DoubleFloat, ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "s17aff",_
  ["x":S,"ifail":S]$Lisp,_
  []$Lisp,_
  [["double":S,"s17affResult":S,"x":S]$Lisp_,
    ["integer":S,"ifail":S]$Lisp_]
  $Lisp_,
  ["s17affResult":S,"ifail":S]$Lisp_,
  [(xArg::Any, ifailArg::Any )]_
  $List Any]$Lisp
  pretend List (Record(key:Symbol, entry:Any))]
$Result

s17agf(xArg:DoubleFloat, ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "s17agf",_
  ["x":S,"ifail":S]$Lisp,_
  []$Lisp,_
  [["double":S,"s17agfResult":S,"x":S]$Lisp_]
  $Lisp_,
  ["s17agfResult":S,"ifail":S]$Lisp_,
  [(xArg::Any, ifailArg::Any )]_
  $List Any]$Lisp
  pretend List (Record(key:Symbol, entry:Any))]
$Result
CHAPTER 15. CHAPTER N

2680

s17ahf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "s17ahf",_
  ["x":S,"ifail":S]$Lisp,_
  [],_AwareList)$List)$List,_
  ["double":S,"s17ahfResult":S,"x"::S]$Lisp,_
  ,["integer":S,"ifail"::S]$Lisp_]
)$Lisp)_

s17ajf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "s17ajf",_
  ["x":S,"ifail":S]$Lisp,_
  [],_AwareList)$List)$List,_
  ["double":S,"s17ajfResult":S,"x"::S]$Lisp,_
  ,["integer":S,"ifail"::S]$Lisp_]
)$Lisp)_

s17akf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "s17akf",_
  ["x":S,"ifail":S]$Lisp,_
  [],_AwareList)$List)$List,_
  ["double":S,"s17akfResult":S,"x"::S]$Lisp,_
  ,["integer":S,"ifail"::S]$Lisp_]
)$Lisp)_

s17dcf(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
  scaleArg:String,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
"s17dcf", [
"fnu":S,"z":S,"n":S,"scale":S,"nz":S,
"ifail":S,"cy":S,"cwrk":S]$Lisp,
"cy":S,"nz":S,"cwrk":S]$Lisp,
["double":S,"fnu":S]$Lisp,
["integer":S,"n":S,"nz":S,"ifail":S]$Lisp,
["character":S,"scale":S]$Lisp,
["double complex":S,"z":S,"cy":S,"n":S]$Lisp,
["cwrk":S,"n":S]$Lisp]$Lisp,
]$Lisp,
["cy":S,"nz":S,"ifail":S]$Lisp,
[(funcall Any,zArg: Any,nArg: Any,scaleArg: Any,ifailArg: Any ])@
pretend List (Record(key:Symbol,entry:Any))]$Result

s17def(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
scaleArg:String,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s17def", [
"fnu":S,"z":S,"n":S,"scale":S,"nz":S,
"ifail":S,"cy":S]$Lisp,
"cy":S,"nz":S]$Lisp,
["double":S,"fnu":S]$Lisp,
["integer":S,"n":S,"nz":S,"ifail":S]$Lisp,
["character":S,"scale":S]$Lisp,
["double complex":S,"z":S,"cy":S,"n":S]$Lisp]$Lisp,
]$Lisp,
["cy":S,"nz":S,"ifail":S]$Lisp,
[(funcall Any,zArg: Any,nArg: Any,scaleArg: Any,ifailArg: Any ])@
pretend List (Record(key:Symbol,entry:Any))]$Result

s17dgf(derivArg:String,zArg:Complex DoubleFloat,scaleArg:String,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s17dgf", 
["deriv":S,"z":S,"scale":S,"ai":S,"nz":S,
"ifail":S]$Lisp,
["ai":S,"nz":S]$Lisp,
["integer":S,"nz":S,"ifail":S]$Lisp,
["character":S,"deriv":S,"scale":S]$Lisp,
["double complex":S,"z":S,"ai":S]$Lisp]$Lisp,
]$Lisp,
["ai":S,"nz":S,"ifail":S]$Lisp,
[(funcall Any,zArg: Any,scaleArg: Any,ifailArg: Any ])@
pretend List (Record(key:Symbol,entry:Any))]$Result

s17dhf(derivArg:String,zArg:Complex DoubleFloat,scaleArg:String,_
ifailArg:Integer): Result ==
[invokeNagman(NIL$Lisp,_
"s17dhf",_
$Lisp,_
["bi":S]$Lisp,_
[["integer":S,"ifail":S]$Lisp_,
["character":S,"deriv":S,"scale":S]$Lisp_,
["double complex":S,"z":S,"bi":S]$Lisp_]
$Lisp_,
["bi":S,"ifail":S]$Lisp_,
[[derivArg::Any,zArg::Any,ifailArg::Any ]]_@
List Any]}.${Lisp).${Lisp)
pretend List (Record(key:Symbol,entry:Any))}$Result

s17dlf(mArg:Integer,fnuArg:DoubleFloat,zArg:Complex DoubleFloat,_
nArg:Integer,-scaleArg::String,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s17dlf",_
["m":S,"fnu":S,"z":S,"n":S,"scale":S_,
"nz":S,"ifail":S,"cy":S]$Lisp_,
["cy":S,"nz":S]$Lisp_,
[["double":S,"fnu":S]$Lisp_,
["integer":S,"m":S,"n":S,"nz":S,"ifail":S_]
$Lisp_,
["character":S,"scale":S]$Lisp_,
["cy":S,"nz":S]$Lisp_,
[[mArg::Any,fnuArg::Any,zArg::Any,nArg::Any,ifailArg::Any ]_@
List Any]}.${Lisp).${Lisp)
pretend List (Record(key:Symbol,entry:Any))}$Result

s18acf(xArg:DoubleFloat,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s18acf",_
["x":S,"ifail":S]$Lisp_,
[])$Lisp_,
[["double":S,"s18acfResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp_,
["s18acfResult":S,"ifail":S]$Lisp_,
[[xArg::Any,ifailArg::Any ]_@
List Any]}.${Lisp).${Lisp)
pretend List (Record(key:Symbol,entry:Any))}$Result

s18adf(xArg:DoubleFloat,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s18adf",_
["x":S,"ifail":S]$Lisp_,
[])$Lisp_,
[["double":S,"s18adfResult":S,"x":S]$Lisp_,
["integer":S,"ifail":S]$Lisp_]
$Lisp_,
["s18adfResult":S,"ifail":S]$Lisp_,
[[xArg::Any,ifailArg::Any ]_@
List Any]}.${Lisp).${Lisp)
pretend List (Record(key:Symbol,entry:Any))}$Result
s18aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_,
    "s18aef",_,
    ["x":S,"ifail":S]$Lisp,_,
    "$Lisp,_
    [["double":S,"s18aefResult":S,"x":S]\$Lisp_ ,
    ["integer":S,"ifail":S]\$Lisp_ ]\$Lisp,_
    ["s18aefResult":S,"ifail":S]\$Lisp,_
    [[[xArg::Any,ifailArg::Any ]]\$Lisp_ @List Any]\$Lisp)$Lisp)_
  pretend List (Record(key:Symbol,entry:Any))]]\Result

s18aff(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_,
    "s18aff",_,
    ["x":S,"ifail":S]$Lisp,_,
    "$Lisp,_
    [["double":S,"s18affResult":S,"x":S]\$Lisp_ ,
    ["integer":S,"ifail":S]\$Lisp_ ]\$Lisp,_
    ["s18affResult":S,"ifail":S]\$Lisp,_
    [[[xArg::Any,ifailArg::Any ]]\$Lisp_ @List Any]\$Lisp)$Lisp)_
  pretend List (Record(key:Symbol,entry:Any))]]\Result

s18dcf(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_,
  scaleArg:String,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_,
    "s18dcf",_,
    "ifail":S,"cy":S]$Lisp,_,
    ["cy":S,"nz":S]\$Lisp,_
    [["double":S,"fnu":S]\$Lisp_ ,
    ["integer":S,"n":S,"nz":S,"ifail":S]\$Lisp_ ,
    ["character":S,"scale":S]\$Lisp_ ,
    ["double complex":S,"z":S,"cy":S,"n":S]\$Lisp]\$Lisp_ ]\$Lisp,_
    ["cy":S,"nz":S,"ifail":S]\$Lisp,_
    [[[fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ]]\$Lisp_ @List Any]\$Lisp)$Lisp)_
s18def(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
  scaleArg:String,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
      "s18def",_  
      ["fnu":S,"z":S,"n":S,"scale":S,"nz":S_  
        ,"ifail":S,"cy":S]$Lisp,_,
      ["cy":S,"nz":S]$Lisp,_,
      [["double":S,"fnu":S]$Lisp_
        ,["character":S,"scale":S]$Lisp_
        ,["double complex":S,"z":S,["cy":S,"n":S]$Lisp]$Lisp_  
      ]$Lisp_,_
      ["cy":S,"nz":S,"ifail":S]$Lisp_,_  
      [[fnuArg::Any,zArg::Any,nArg::Any,ifailArg::Any ]]_  
    @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s19aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
      "s19aaf",_  
      ["x":S,"ifail":S]$Lisp,_,
    ]$Lisp,_,
      ["double":S,"s19aafResult":S,"x":S]$Lisp_  
      ,["integer":S,"ifail":S]$Lisp_  
      ]$Lisp_,_
    ["s19aafResult":S,"ifail":S]$Lisp_,_  
    [[xArg::Any,ifailArg::Any ]]_  
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s19abf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
      "s19abf",_  
      ["x":S,"ifail":S]$Lisp,_,
    ]$Lisp,_,
      ["double":S,"s19abfResult":S,"x":S]$Lisp_  
      ,["integer":S,"ifail":S]$Lisp_  
      ]$Lisp_,_
    ["s19abfResult":S,"ifail":S]$Lisp_,_  
    [[xArg::Any,ifailArg::Any ]]_  
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s19acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [[(invokeNagman(NIL$Lisp,_
      "s19acf",_  
      ["x":S,"ifail":S]$Lisp,_,
    ]$Lisp,_,
      ["double":S,"s19acfResult":S,"x":S]$Lisp_  
      ,["integer":S,"ifail":S]$Lisp_  
      ]$Lisp_,_
    ["s19acfResult":S,"ifail":S]$Lisp_,_  
    [[xArg::Any,ifailArg::Any ]]_  
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result
s19adf(xArg:DoubleFloat,ifailArg:Integer): Result ==

[(invokeNagman(NIL$Lisp,_,
  "s19adf",_,
  ["x"::S,"ifail"::S]$Lisp_,
  []$Lisp_,
  [["double"::S,"s19adfResult"::S,"x"::S]$Lisp_,
    [["integer"::S,"ifail"::S]$Lisp_]
  ]$Lisp_,
  ["s19adfResult"::S,"ifail"::S]$Lisp_,
  [(xArg::Any,ifailArg::Any )]
  @List Any]$Lisp)$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s20acf(xArg:DoubleFloat,ifailArg:Integer): Result ==

[(invokeNagman(NIL$Lisp,_,
  "s20acf",_,
  ["x"::S,"ifail"::S]$Lisp_,
  []$Lisp_,
  [["double"::S,"s20acfResult"::S,"x"::S]$Lisp_,
    [["integer"::S,"ifail"::S]$Lisp_]
  ]$Lisp_,
  ["s20acfResult"::S,"ifail"::S]$Lisp_,
  [(xArg::Any,ifailArg::Any )]
  @List Any]$Lisp)$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s20adf(xArg:DoubleFloat,ifailArg:Integer): Result ==

[(invokeNagman(NIL$Lisp,_,
  "s20adf",_,
  ["x"::S,"ifail"::S]$Lisp_,
  []$Lisp_,
  [["double"::S,"s20adfResult"::S,"x"::S]$Lisp_,
    [["integer"::S,"ifail"::S]$Lisp_]
  ]$Lisp_,
  ["s20adfResult"::S,"ifail"::S]$Lisp_,
  [(xArg::Any,ifailArg::Any )]
  @List Any]$Lisp)$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s21baf(xArg:DoubleFloat,yArg:DoubleFloat,ifailArg:Integer): Result ==

[(invokeNagman(NIL$Lisp,_,
  "s21baf",_
CHAPTER 15. CHAPTER N

Lisp, 
Lisp, 
Lisp_

"double"::S,"s21bafResult"::S,"x"::S,"y"::S_ 
[Lisp_ 
, ["integer"::S,"ifail"::S]$Lisp_
$Lisp_ 
"s21bafResult"::S,"ifail"::S]$Lisp_, 
[(xArg::Any,yArg::Any,ifailArg::Any )_ 
@List Any][$Lisp)[$Lisp) 
pretend List (Record(key:Symbol,entry:Any))]

Result 

s21bbf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,_ 
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp_, 
"s21bbf", 
["x":S,"y":S,"z":S,"ifail":S]$Lisp_, 
[])$Lisp_, 
["double":S,"s21bbfResult"::S,"x"::S,"y"::S_ 
,"z"::S]$Lisp_
 , ["integer"::S,"ifail"::S]$Lisp_ 
$Lisp_, 
"s21bbfResult"::S,"ifail"::S]$Lisp_, 
[(xArg::Any,yArg::Any,zArg::Any,ifailArg::Any )_ 
@List Any][$Lisp)[$Lisp) 
pretend List (Record(key:Symbol,entry:Any))]

Result 

s21bcf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,_ 
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp_, 
"s21bcf", 
["x":S,"y":S,"z":S,"ifail":S]$Lisp_, 
[])$Lisp_, 
["double":S,"s21bcfResult"::S,"x"::S,"y"::S_ 
,"z"::S]$Lisp_
 , ["integer"::S,"ifail"::S]$Lisp_ 
$Lisp_, 
"s21bcfResult"::S,"ifail"::S]$Lisp_, 
[(xArg::Any,yArg::Any,zArg::Any,ifailArg::Any )_ 
@List Any][$Lisp)[$Lisp) 
pretend List (Record(key:Symbol,entry:Any))]

Result 

s21bdf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,rArg:DoubleFloat,_ 
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp_, 
"s21bdf", 
["x":S,"y":S,"z":S,"r":S,"ifail":S_ 
]$Lisp_, 
[])$Lisp_, 
["double":S,"s21bdfResult"::S,"x"::S,"y"::S_ 
,"z"::S,"r"::S]$Lisp_
package NSUP2 NewSparseUnivariatePolynomialFunctions2

-- S 1 of 1
)show NewSparseUnivariatePolynomialFunctions2
-- E 1

)lisp (bye)
This package lifts a mapping from coefficient rings $R$ to $S$ to a mapping from sparse univariate polynomial over $R$ to a sparse univariate polynomial over $S$. Note that the mapping is assumed to send zero to zero, since it will only be applied to the non-zero coefficients of the polynomial.

See Also:
- )show NewSparseUnivariatePolynomialFunctions2

---

**NewSparseUnivariatePolynomialFunctions2** (NSUP2)

Exports:
- `map`

---

)abbrev package NSUP2 NewSparseUnivariatePolynomialFunctions2
++ Description:
++ This package lifts a mapping from coefficient rings $R$ to $S$ to
++ a mapping from sparse univariate polynomial over $R$ to
++ a sparse univariate polynomial over $S$.
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

NewSparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
map:(R->S,NewSparseUnivariatePolynomial R) -> NewSparseUnivariatePolynomial S
++ \axiom{map(func, poly)} creates a new polynomial by applying func to
++ every non-zero coefficient of the polynomial poly.
== add
map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
This package exports Newton interpolation for the special case where the result is known to be in the original integral domain. The packages defined in this file provide fast fraction free rational interpolation.
NewtonInterpolation (NEWTON)

Exports:
newton

— package NEWTON NewtonInterpolation —

)abbrev package NEWTON NewtonInterpolation
++ Description:
++ This package exports Newton interpolation for the special case where the
++ result is known to be in the original integral domain
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

NewtonInterpolation F: Exports == Implementation where
F: IntegralDomain
Exports == with

newton: List F -> SparseUnivariatePolynomial F

++ \spad{newton}(l) returns the interpolating polynomial for the values
++ l, where the x-coordinates are assumed to be [1,2,3,...,n] and the
++ coefficients of the interpolating polynomial are known to be in the
++ domain F. I.e., it is a very streamlined version for a special case of
++ interpolation.
Implementation == add

\[
\text{differences(yl: List F): List F ==}
\]
\[
[y2-y1 for y1 in yl for y2 in rest yl]
\]

\[
z: \text{SparseUnivariatePolynomial(F) := monomial(1,1)}
\]

-- we assume x=[1,2,3,...,n]

newtonAux(k: F, fact: F, yl: List F): SparseUnivariatePolynomial(F) ==

if empty? rest yl
    then ((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
else (yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
    + (z-k::SparseUnivariatePolynomial(F)) _
    * newtonAux(k+1$F, fact*k, differences yl)

newton yl == newtonAux(1$F, 1$F, yl)
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NPOLYGON
--R
--R------------------------------- Operations --------------------------------
--R multiplicity : List(List(PolyRing)) -> NonNegativeInteger
--R negAndPosEdge : (PolyRing,List(List(PolyRing))) -> List(List(PolyRing))
--R newtonPolygon : (PolyRing,Integer,Integer,Union(left,center,right,vertical,horizontal)) -> List(List(PolyRing))
--R slope : (PolyRing,PolyRing) -> Record(height: Integer,base: Integer,quotient: Integer,reste: Integer,type: Union(left,center,right,vertical,horizontal))
--R slope : List(PolyRing) -> Record(height: Integer,base: Integer,quotient: Integer,reste: Integer,type: Union(left,center,right,vertical,horizontal))
--R
--E 1

)spool
)lisp (bye)

——

---------- NewtonPolygon.help
——

====================================================================
NewtonPolygon examples
====================================================================

The following is part of the PAFF package

See Also:
o )show NewtonPolygon

——

NewtonPolygon (N POLYGON)
package NPOLYGON NewtonPolygon

)abbrev package NPOLYGON NewtonPolygon
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
++ NewtonPolygon(K,PolyRing,E,dim):Exports == Implementation where
++ K:Ring
++ dim:NonNegativeInteger
++ PolyRing: FiniteAbelianMonoidRing(K,E)
++ E: DirectProductCategory(dim,NonNegativeInteger)

PackPoly ==> PackageForPoly(K,PolyRing,E,dim)

recSlope ==> Record( height:Integer, base:Integer, quotient:Integer, _
  reste:Integer, _
  type:Union("left","center","right","vertical","horizontal"))

Exports ==> with

  newtonPolygon: (PolyRing,Integer,Integer,Union("left","center", "right",_
  "vertical","horizontal")) -> List List PolyRing

  multiplicity: List List PolyRing -> NonNegativeInteger

  negAndPosEdge: (PolyRing, List List PolyRing) -> List List PolyRing

  slope: (PolyRing,PolyRing) -> recSlope
  slope: List PolyRing -> recSlope

Implementation ==> add

slope(p1,p2) ==
  -- calcule la pente de p1 a p2 et change le signe.
  e1:=degree p1
e2:=degree p2
  hgt:=( e1.2 pretend Integer) - ( e2.2 pretend Integer)
  bs:=( e2.1 pretend Integer) - ( e1.1 pretend Integer)
  zero? bs => [hgt, bs, 0$Integer, 0$Integer, "vertical"]$recSlope
  zero? hgt => [hgt, bs, 0$Integer, 0$Integer, "horizontal"]$recSlope
  hgt = bs => [hgt, bs, 1$Integer, 0$Integer, "center"]$recSlope
  hgt > bs =>
    eucl:=divide(hgt,bs)
    [hgt, bs, eucl.quotient, eucl.remainder , "left"]$recSlope
    eucl:=divide(bs, hgt)
    [hgt, bs, eucl.quotient, eucl.remainder , "right"]$recSlope
oneToPos:: List List PolyRing -> List List PolyRing

oneToPos(lpol)==
  fedge:= first lpol
  sl:= slope fedge
  one? ( #(lpol) ) =>
    if sl.height > sl.base then [ fedge, empty() ]
    else [ empty(), fedge ]
  ^( sl.base < sl.height ) => [ empty(), fedge ]
  restPANE:= oneToPos rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ^( sl2.base < sl2.height ) => [ fedge , fedge2 ]
  restPANE

oneToNeg:: List List PolyRing -> List List PolyRing

oneToNeg(lpol)==
  fedge:= first lpol
  sl:= slope fedge
  one? ( #(lpol) ) =>
    if sl.height < sl.base then [ empty(), fedge ]
    else [ fedge, empty() ]
  ^( sl.height < sl.base ) => [ empty(), fedge ]
  restPANE:= oneToNeg rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ^( sl2.height < sl2.base ) => [ fedge , fedge2 ]
  restPANE

negAndPosEdge(pol, lpol)==
  -- cette fonction retourne deux liste de polynomes:
  -- la premiere est liee a
  -- la transformation x = x y^l (i.e v(x) >= v(y) ).
  -- la deuxieme est liee a la transformation
  -- y = x^l y (i.e. v(x) <= v(y) ).
  -- si le degree en Y est inferieur a celui en X on
  -- previligie la transformation
  -- y = x^l y.
  degree( pol , 2 )$PackPoly < degree( pol, 1 )$PackPoly => oneToPos lpol
  oneToNeg lpol
  oneToNeg lpol

localNewtonPolygon: List PolyRing -> List PolyRing

slEq: (recSlope, recSlope) -> Boolean

regroup: List PolyRing -> List List PolyRing

multiplicity( lpol )==
  nl:=#(lpol)
flpol := first lpol
one? nl => totalDegree( last flpol )$PackPoly
s := slope flpol
s.height < s.base => totalDegree( first flpol )$PackPoly
multiplicity( rest lpol )

slEq(s1, s2) ==
s1.height * s2.base = s2.height * s1.base

regroup(lpol) ==
 -- Note : les elements de lpol sont sur la frontiere d'un poly.
 -- de Newton et il sont deja trie's.
nl := #(lpol)
one? nl => [lpol]
2 = nl => [lpol]
f := first lpol
r := regroup rest lpol
-- Note : les listes de "r" contiennent au moins 2 elements !!
f := first r
s1 := slope(f, first fg)
s2 := slope(fg.1, fg.2)
slEq(s1, s2) => cons( cons(f, fg) , rest r)
cons( [f, first fg], r)

-- ==============================================================
-- sortMono : trie les monomes par ordre croissant
-- ==============================================================

sortMono: (PolyRing, PolyRing) -> Boolean
sortMono(p1, p2) ==
a := degree p1
b := degree p2
a.1 < b.1 => true -- p1 est a gauche de p2
a.1 = b.1 and a.2 > b.2 => true -- p1 est au dessus de p2
false

-- ==============================================================
-- newtonPolygon : retourne tous les monomes sur la
-- frontiere de du polygone de Newton,
-- regroupes selon leur pente.
-- ==============================================================

properSlope: ( List PolyRing, Integer, Integer, 
        Union("left","center","right","vertical","horizontal")) -> Boolean

properSlope(lpol, hgt, bs, tp) ==
s := slope lpol
tp case "left" and s.height = hgt and s.base = bs => true
tp case "right" and s.height = bs and s.base = hgt => true
false
newtonPolygon(pol, hgt, bs, tp) == 
   ans := regroup localNewtonPolygon _
           sort( sortMono(#1, #2), monomials(pol)$PackPoly)
   zero?(bs) => ans
   [ l for l in ans | properSlope(l, hgt, bs, tp)]

comp2pol: (PolyRing, PolyRing) -> List PolyRing
comp2pol(p1, p2) ==
   rs := slope(p1, p2)
   zero? rs.base => -- p1 et p2 sont alignes verticalement !!
   zero? rs.height => [p1 + p2] -- les monomes sont identiques !
   rs.height < 0 => [p1] -- p2 est au dessus de p1,
   rs.base > 0 => -- p1 est a gauche de p2
   rs.height > 0 => [p1, p2] -- p1 est plus haut que p2
   rs.height < 0 => [p2, p1] -- p2 est plus haut que p1
   [p2] -- p2 est a la meme hauteur que p1.

slope(lpol) ==
   ~one?(#lpol) => slope( first lpol, second lpol)
   f := first lpol
   ( degree(f, 2)$PackPoly < degree(f, 1)$PackPoly ) => _
      [0$Integer, 1$Integer, 0$Integer, 0$Integer, "right"]$recSlope
      [1$Integer, 0$Integer, 0$Integer, 0$Integer, "left"]$recSlope

convex_?: (PolyRing, PolyRing, PolyRing) -> Boolean
convex_?(p1, p2, p3) ==
   s1 := slope(p1, p2)
   s2 := slope(p2, p3)
   s1.type case "horizontal" => true
   s2.type case "vertical" => true
   s1.type case "vertical" => false -- car ici il faut c2 vertical
   s2.type case "horizontal" => false
   (s1.height * s2.base) < (s2.height * s1.base)

consBondary: (PolyRing, List PolyRing) -> List PolyRing
consBondary(lt, lpol) ==
   -- "lt" est un monome a ajouter ou non a "lpol" qui est une
   empty? lpol => [lt]
   st := first lpol
   n1 := NonNegativeInteger: # lpol
   one? n1 => comp2pol(lt, st)
   degree(lt).1 = degree(st).1 and degree(lt).2 > degree(st).2 => lpol
   ~convex?(lt, st, lpol.2) => cons(lt, lpol)
   consBondary( lt, rest lpol )
localNewtonPolygon(lpol)==
  -- lpol doit etre trie' par sortMono
  empty? lpol => empty()
  nl:= #(lpol)
  one? nl => lpol
  lt:=first lpol
  polgRest:= localNewtonPolygon rest lpol
  consBondary( lt, polgRest )

— NPOLYGON.dotabb —

"NPOLYGON" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPOLYGON"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"NPOLYGON" -> "DIRPCAT"

— NonCommutativeOperatorDivision.input —

)set break resume
)sys rm -f NonCommutativeOperatorDivision.output
)spool NonCommutativeOperatorDivision.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NonCommutativeOperatorDivision
--E 1

)spool
)lisp (bye)

— NonCommutativeOperatorDivision.help —
NonCommutativeOperatorDivision examples
====================================================================

This package provides a division and related operations for
MonogenicLinearOperator over a Field.
Since the multiplication is in general non-commutative,
these operations all have left- and right-hand versions.
This package provides the operations based on left-division.

\[ q, r \] = leftDivide(a, b) means \( a = b \times q + r \)

See Also:
o )show NonCommutativeOperatorDivision

NonCommutativeOperatorDivision (NCODIV)

Exports:
leftGcd  leftLcm  leftQuotient  leftRemainder  leftDivide  leftExactQuotient

--- package NCODIV NonCommutativeOperatorDivision ---

)abbrev package NCODIV NonCommutativeOperatorDivision
++ Author: Jean Della Dora, Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides a division and related operations for
++ \spadtype{MonogenicLinearOperator}s over a \spadtype{Field}.
++ Since the multiplication is in general non-commutative,
++ these operations all have left- and right-hand versions.
++ This package provides the operations based on left-division.\br
++ \tab{5}[q,r] = leftDivide(a,b) means \(a=b\cdot q+r\)

NonCommutativeOperatorDivision(P, F): PDcat == PDdef where
  P: MonogenicLinearOperator(F)
  F: Field

PDcat == with
  leftDivide: (P, P) -> Record(quotient: P, remainder: P)
    ++ \spad{\text{leftDivide}(a,b)} returns the pair \spad{[q,r]} such that
    ++ \spad{a = b\cdot q + r} and the degree of \spad{r} is
    ++ less than the degree of \spad{b}.
    ++ This process is called `left division`.
  leftQuotient: (P, P) -> P
    ++ \spad{\text{leftQuotient}(a,b)} computes the pair \spad{[q,r]} such that
    ++ \spad{a = b\cdot q + r} and the degree of \spad{r} is
    ++ less than the degree of \spad{b}.
    ++ The value \spad{\text{leftQuotient}(a,b)} is returned.
  leftRemainder: (P, P) -> P
    ++ \spad{\text{leftRemainder}(a,b)} computes the pair \spad{[q,r]} such that
    ++ \spad{a = b\cdot q + r} and the degree of \spad{r} is
    ++ less than the degree of \spad{b}.
    ++ The value \spad{\text{leftRemainder}(a,b)} is returned.
  leftExactQuotient: (P, P) -> Union(P, "failed")
    ++ \spad{\text{leftExactQuotient}(a,b)} computes the value \spad{\text{leftQuotient}(a,b)} if it exists,
    ++ such that \spad{\text{leftQuotient}(a,b) = a = b\cdot q}.
  leftGcd: (P, P) -> P
    ++ \spad{\text{leftGcd}(a,b)} computes the value \spad{\text{leftQuotient}(a,b)} of highest degree
    ++ such that
    ++ \spad{\text{leftQuotient}(a,b) = a = a\cdot g}
    ++ \spad{\text{leftQuotient}(a,b) = b = b\cdot g}
    ++ for some values \spad{\text{leftQuotient}(a,b) = a\cdot g} and \spad{\text{leftQuotient}(a,b) = b\cdot g}.
    ++ The value \spad{\text{leftQuotient}(a,b) = a = b\cdot g} is computed using \spad{\text{leftDivision}}.
  leftLcm: (P, P) -> P
    ++ \spad{\text{leftLcm}(a,b)} computes the value \spad{\text{leftQuotient}(a,b)} of lowest degree
    ++ such that \spad{\text{leftQuotient}(a,b) = a = a\cdot g = b\cdot b\cdot g} for some values
    ++ \spad{\text{leftQuotient}(a,b) = a = a\cdot g} and \spad{\text{leftQuotient}(a,b) = b = b\cdot g}.
    ++ The value \spad{\text{leftQuotient}(a,b) = a = b\cdot g} is computed using \spad{\text{leftDivision}}.

PDdef == add
  leftDivide(a, b) ==
    q: P := 0
    r: P := a
    iv:F := inv leadingCoefficient b
    while degree r >= degree b and r ^= 0 repeat
      h := monomial(iv*leadingCoefficient r,
        (degree r - degree b)::NonNegativeInteger)$P
        r := r - b*h
      q := q + h
    [q,r]
-- leftQuotient(a,b) is the quotient from left division, etc.
leftQuotient(a,b) == leftDivide(a,b).quotient
leftRemainder(a,b) == leftDivide(a,b).remainder
leftExactQuotient(a,b) ==
  qr := leftDivide(a,b)
  if qr.remainder = 0 then qr.quotient else "failed"

-- l = leftGcd(a,b) means a = aa*l b = bb*l. Uses leftDivide.
leftGcd(a,b) ==
a = 0 => b
b = 0 => a
while degree b > 0 repeat (a,b) := (b, leftRemainder(a,b))
if b=0 then a else b

-- l = leftLcm(a,b) means l = a*aa  l = b*bb  Uses leftDivide.
leftLcm(a,b) ==
a = 0 => b
b = 0 => a
b0 := b
u := monomial(1,0)$P
v := 0
while leadingCoefficient b ^= 0 repeat
  qr := leftDivide(a,b)
  (a, b) := (b, qr.remainder)
  (u, v) := (u*qr.quotient+v, u)
b0*u

— NCODIV.dotabb —

"NCODIV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCODIV"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"NCODIV" -> "FIELD"

package NONE1 NoneFunctions1

— NoneFunctions1.input —

)set break resume
)sys rm -f NoneFunctions1.output
)spool NoneFunctions1.output
)set message test on


NoneFunctions1 implements functions on None. It particular it includes a particulary dangerous coercion from any other type to \spadtype{None}.

See Also:

- o )show NoneFunctions1

---

NoneFunctions1 (NONE1)

Exports:

- coerce

---
\texttt{NoneFunctions1(S:Type): Exports == Implementation where}
\begin{verbatim}
Exports ==> with
coerce: S -> None
++ coerce(x) changes \spad{x} into an object of type
++ \spad{None}.

Implementation ==> add
coerce(s:S):None == s pretend None
\end{verbatim}
NonLinearFirstOrderODESolver provides a function for finding closed form first integrals of nonlinear ordinary differential equations of order 1.

See Also:
o )show NonLinearFirstOrderODESolver

NonLinearFirstOrderODESolver (NODE1)

Exports:
solve

)abbrev package NODE1 NonLinearFirstOrderODESolver
++ Author: Manuel Bronstein
++ Date Created: 2 September 1991
++ Date Last Updated: 14 October 1994
++ Description:
++ NonLinearFirstOrderODESolver provides a function
++ for finding closed form first integrals of nonlinear ordinary
++ differential equations of order 1.

NonLinearFirstOrderODESolver(R, F):Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer, LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory, PrimitiveFunctionCategory)

N ==> NonNegativeInteger
Q ==> Fraction Integer
UQ ==> Union(Q, "failed")
OP ==> BasicOperator
SY ==> Symbol
K ==> Kernel F
U ==> Union(F, "failed")
P ==> SparseMultivariatePolynomial(R, K)
REC ==> Record(coef:Q, logand:F)
SOL ==> Record(particular: F, basis: List F)
BER ==> Record(coef1:F, coefn:F, exponent:N)

Exports ==> with
solve: (F, F, OP, SY) -> U
++ solve(M(x,y), N(x,y), y, x) returns \spad{F(x,y)} such that
++ \spad{F(x,y) = c} for a constant \spad{c} is a first integral
++ of the equation \spad{M(x,y) dx + N(x,y) dy = 0}, or
++ "failed" if no first-integral can be found.

Implementation ==> add
import ODEIntegration(R, F)
import ElementaryFunctionODESolver(R, F) -- recursive dependency!

checkBernoulli : (F, F, K) -> Union(BER, "failed")
solveBernoulli : (BER, OP, SY, F) -> Union(F, "failed")
checkRiccati : (F, F, K) -> Union(List F, "failed")
solveRiccati : (List F, OP, SY, F) -> Union(F, "failed")
partSolRiccati : (List F, OP, SY, F) -> Union(F, "failed")
integratingFactor: (F, F, SY, SY) -> U

unk := new()$SY
kunk:K := kernel unk

solve(m, n, y, x) ==
-- first replace the operator y(x) by a new symbol z in m(x,y) and n(x,y)
lk:List(K) := [retract(yx := y(x::F))@K]
lv:List(F) := [kunk::F]
mm := eval(m, lk, lv)
nn := eval(n, lk, lv)
-- put over a common denominator (to balance m and n)
d := lcm(denom mm, denom nn)::F
mm := d * mm
nn := d * nn
-- look for an integrating factor mu
(u := integratingFactor(mm, nn, unk, x)) case F =>
mu := u::F
mm := mm * mu
nn := nn * mu
eval(int(mm, x) + int(nn-int(differentiate(mm, unk), x), unk), [kunk], [yx])
-- check for Bernoulli equation
(w := checkBernoulli(m, n, k1 := first lk)) case BER =>
solveBernoulli(w::BER, y, x, yx)
-- check for Riccati equation
(v := checkRiccati(m, n, k1)) case List(F) =>
solveRiccati(v::List(F), y, x, yx)
"failed"

-- look for an integrating factor
integratingFactor(m, n, y, x) ==
-- check first for exactness
zero?(d := differentiate(m, y) - differentiate(n, x)) => 1
-- look for an integrating factor involving x only
not member?(y, variables(f := d / n)) => expint(f, x)
-- look for an integrating factor involving y only
not member?(x, variables(f := - d / m)) => expint(f, y)
-- room for more techniques later on (e.g. Prelle-Singer etc...)
"failed"

-- check whether the equation is of the form
-- dy/dx + p(x)y + q(x)y^N = 0 with N > 1
-- i.e. whether m/n is of the form p(x) y + q(x) y^N
-- returns [p, q, N] if the equation is in that form
checkBernoulli(m, n, ky) ==
r := denom(f := m / n)::F
(not freeOf?(r, y := ky::F))
or (d := degree(p := univariate(numer f, ky))) < 2
or degree(pp := reductum p) ^= 1 or reductum(pp) ^= 0
or (not freeOf?(a := (leadingCoefficient(pp)::F), y))
or (not freeOf?(b := (leadingCoefficient(p)::F), y)) => "failed"
[a / r, b / r, d]

-- solves the equation dy/dx + rec.coef1 y + rec.coefn y^rec.exponent = 0
-- the change of variable v = y^{1-n} transforms the above equation to
-- dv/dx + (1 - n) p v + (1 - n) q = 0
solveBernoulli(rec, y, x, yx) ==
n1 := 1 - rec.exponent::Integer
deq := differentiate(yx, x) + n1 * rec.coef1 * yx + n1 * rec.coefn
sol := solve(deq, y, x)::SOL
-- can always solve for order 1
-- if v = vp + c v0 is the general solution of the linear equation, then
-- the general first integral for the Bernoulli equation is
-- (y^{1-n} - vp) / v0 = c for any constant c
(yx**n1 - sol.particular) / first(sol.basis)

-- check whether the equation is of the form
-- dy/dx + q0(x) y + q1(x)y + q2(x)y^2 = 0
-- i.e. whether \( m/n \) is a quadratic polynomial in \( y \).
-- returns the list \([q_0, q_1, q_2]\) if the equation is in that form
\[
\text{checkRiccati}(m, n, ky) ==
\]
\[
q := \text{denom}(f := m / n)::F
\]
\[
(\text{not freeOf?(}q, y := ky::F) \lor \text{degree(p := univariate(numer f, ky)) > 2})
\]
\[
\text{or (not freeOf?(a0 := (coefficient(p, 0)::F), y))}
\]
\[
\text{or (not freeOf?(a1 := (coefficient(p, 1)::F), y))}
\]
\[
\text{or (not freeOf?(a2 := (coefficient(p, 2)::F), y))} \Rightarrow "failed"
\]
\[
[a0 / q, a1 / q, a2 / q]
\]
-- solves the equation \( \frac{dy}{dx} + 1.1 + 1.2 y + 1.3 y^2 = 0 \)
\[
\text{solveRiccati}(1, y, x, yx) ==
\]
-- get first a particular solution
\[
(u := \text{partSolRiccati}(1, y, x, yx)) \text{ case "failed" } \Rightarrow "failed"
\]
-- once a particular solution \( y_p \) is known, the general solution is of the form \( y = y_p + 1/v \) where \( v \) satisfies the linear 1st order equation
-- \( v' - (1.2 + 2 \times 1.3 y_p) v = 1.3 \)
\[
deq := \text{differentiate(yx, x)} - (1.2 + 2 \times 1.3 \times u::F) \times yx - 1.3
\]
\[
gsol := \text{solve}(\text{deq}, y, x)::\text{SOL}
\]
\[-\text{can always solve for order 1}
\]
-- if \( v = v_p + c v_0 \) is the general solution of the above equation, then
-- the general first integral for the Riccati equation is
-- \( \frac{1/(y - y_p) - v_0}{v_0} = c \) for any constant \( c \)
\[
(\text{inv(yx - u::F) - gsol.particular}) / \text{first(gsol.basis)}
\]
-- looks for a particular solution of \( \frac{dy}{dx} + 1.1 + 1.2 y + 1.3 y^2 = 0 \)
\[
\text{partSolRiccati}(1, y, x, yx) ==
\]
-- we first do the change of variable \( y = z / 1.3 \), which transforms
-- the equation into \( \frac{dz}{dx} + 1.1 1.3 + (1.2 - 1.3' / 1.3) z + z^2 = 0 \)
\[
q0 := 1.1 \ast (13 := 1.3)
\]
\[
q1 := 1.2 - \text{differentiate}(13, x) / 13
\]
-- the equation \( \frac{dz}{dx} + q0 + q1 z + z^2 = 0 \) is transformed by the change
-- of variable \( z = w'/w \) into the linear equation \( w'' + q1 \ w' + q0 \ w = 0 \)
\[
\text{lineq := differentiate(yx, x) + q1 \text{ differentiate(yx, x)} + q0 \text{ yx}}
\]
-- should be made faster by requesting a particular nonzero solution only
\[
(\text{not((gsol := solve(lineq, y, x)) case SOL}))
\]
\[
\text{or empty?(bas := (gsol::SOL).basis)} \Rightarrow "failed"
\]
\[
\text{differentiate(first bas, x)} / (13 \ast \text{first bas})
\]
---

--- NODE1.dotabb ---

"NODE1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NODE1"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"NODE1" -> "ACFS"
package NLINSOL NonLinearSolvePackage

-- NonLinearSolvePackage.input --

)set break resume
)sys rm -f NonLinearSolvePackage.out
)spool NonLinearSolvePackage.out
)set message test on
)set message auto off
)clear all

--) S 1 of 1
)show NonLinearSolvePackage
--) E 1

)spool
)lisp (bye)

---

-- NonLinearSolvePackage.help --

===============================================
NonLinearSolvePackage examples
===============================================

NonLinearSolvePackage is an interface to SystemSolvePackage that attempts to retract the coefficients of the equations before solving. The solutions are given in the algebraic closure of \( \mathbb{R} \) whenever possible.

See Also:
o )show NonLinearSolvePackage

---
NonLinearSolvePackage (NLINSOL)

Exports:
solve   solveInField

— package NLINSOL NonLinearSolvePackage —

abbrev package NLINSOL NonLinearSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 26 June 1992
++ Description:
++ NonLinearSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving. The solutions are given in the algebraic closure of \( R \) whenever
++ possible.

NonLinearSolvePackage(R: IntegralDomain): Exports == Implementation where
Z   ==> Integer
Q   ==> Fraction Z
SY  ==> Symbol
P   ==> Polynomial R
F   ==> Fraction P
EQ  ==> Equation F
SSP ==> SystemSolvePackage
SOL ==> RetractSolvePackage

Exports => with
solveInField: (List P, List SY) -> List List EQ
++ solveInField(lp,lv) finds the solutions of the list lp of
++ rational functions with respect to the list of symbols lv.
solveInField: List P   -> List List EQ
++ solveInField(lp) finds the solution of the list lp of rational
++ functions with respect to all the symbols appearing in lp.
solve: (List P, List SY) -> List List EQ
++ solve(lp,lv) finds the solutions in the algebraic closure of \( R \)
++ of the list lp of
++ rational functions with respect to the list of symbols lv.
solve: List P -> List List EQ
++ solve(lp) finds the solution in the algebraic closure of R
++ of the list lp of rational
++ functions with respect to all the symbols appearing in lp.

Implementation ==> add
solveInField l == solveInField(l, "setUnion"/[variables p for p in l])

if R has AlgebraicallyClosedField then
  import RationalFunction(R)

expandSol: List EQ -> List List EQ
RIfCan : F -> Union(R, "failed")
addRoot : (EQ, List List EQ) -> List List EQ
allRoots : List P -> List List EQ
evalSol : (List EQ, List EQ) -> List EQ

solve l == solve(l, "setUnion"/[variables p for p in l])
solve(lp, lv) == concat([expandSol sol for sol in solveInField(lp, lv)])
addRoot(eq, l) == [concat(eq, sol) for sol in l]
evalSol(ls, l) == [equation(lhs eq, eval(rhs eq, l)) for eq in ls]

-- converts [p1(a1),...,pn(an)] to
-- [[a1=v1,...,an=vn]] where vi ranges over all the zeros of pi
allRoots l ==
  empty? l => [empty()$List(EQ)]
  z := allRoots rest l
  s := mainVariable(p := first l)::SY::P::F
  concat [addRoot(equation(s, a::P::F), z) for a in zerosOf univariate p]

expandSol l ==
  lassign := lsubs := empty()$List(EQ)
  luniv := empty()$List(P)
  for eq in l repeat
    if retractIfCan(lhs eq)@Union(SY, "failed") case SY then
      if RIfCan(rhs eq) case R then lassign := concat(eq, lassign)
        else lsubs := concat(eq, lsubs)
      else
        if ((u := retractIfCan(lhs eq)@Union(P, "failed")) case P) and
        one?(# variables(u::P)) and (r := RIfCan rhs eq) case R then
          luniv := concat(u::P - r::R::P, luniv)
        else return [l]
    empty? luniv => [l]
    [concat(z, concat(evalSol(lsubs, z), lassign)) for z in allRoots luniv]

RIfCan f ==
  ((n := retractIfCan(numer f)@Union(R, "failed")) case R) and
  ((d := retractIfCan(denom f)@Union(R, "failed")) case R) => n::R / d::R
"failed"
else
  solve l == solveInField l
  solve(lp, lv) == solveInField(lp, lv)

-- 'else if' is doubtful with this compiler so all 3 conditions are explicit
if (not(R is Q)) and (R has RetractableTo Q) then
  solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Q, R)

if (not(R is Z)) and (not(R has RetractableTo Q)) and
  (R has RetractableTo Z) then
  solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Z, R)

if (not(R is Z)) and (not(R has RetractableTo Q)) and
  (not(R has RetractableTo Z)) then
  solveInField(lp, lv) == solve([p::F for p in lp]$List(F), lv)$SSP(R)

---

-- NLINSOL.dotabb --

"NLINSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NLINSOL"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"NLINSOL" -> "ACF"

---

package NORMPK NormalizationPackage

--- NormalizationPackage.input ---

)set break resume
)sys rm -f NormalizationPackage.output
)spool NormalizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NormalizationPackage
--E 1

)spool
)lisp (bye)
— NormalizationPackage.help —

NormalizationPackage examples

A package for computing normalized associates of univariate polynomials with coefficients in a tower of simple extensions of a field.

See Also:
- )show NormalizationPackage

NormalizationPackage (NORMPK)

Exports:
- normalizedAssociate
- normInvertible?
- normalize
- outputArgs
- recip

— package NORMPK NormalizationPackage —
A package for computing normalized associates of univariate polynomials with coefficients in a tower of simple extensions of a field.

NormalizationPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Split ==> List TS
--KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
--EntryGcd ==> List PWT
--HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
--KeyInvSet ==> Record(arg1: P, arg3: TS)
--EntryInvSet ==> List TS
--HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

recip: (P, TS) -> Record(num:P,den:P)
++ \texttt{recip(p,ts)} returns the inverse of \texttt{p} w.r.t \texttt{spad{ts}}
++ assuming that \texttt{p} is invertible w.r.t \texttt{spad{ts}}.

normalizedAssociate: (P, TS) -> P
++ \texttt{normalizedAssociate(p,ts)} returns a normalized polynomial
++ \texttt{spad{ts}} such that \texttt{p} and \texttt{normalizedAssociate(p,ts)} are
++ associates w.r.t \texttt{spad{ts}} and assuming that \texttt{p} is invertible
++ w.r.t \texttt{spad{ts}}.

normalize: (P, TS) -> List PWT
++ \texttt{normalize(p,ts)} normalizes \texttt{p} w.r.t \texttt{spad{ts}}.

outputArgs: (S, S, P, TS) -> Void
++ \texttt{outputArgs(s1,s2,p,ts)}
++ is an internal subroutine, exported only for development.

normInvertible?: (P, TS) -> List BWT
++ \texttt{normInvertible?(p,ts)}
++ is an internal subroutine, exported only for development.

Implementation == add

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P) then

\texttt{normInvertible?}(p:P, ts:TS): List BWT ==
\texttt{stoneInvertible?}_sqfreg(p,ts)$regsetgcdpack

else

\texttt{normInvertible?}(p:P, ts:TS): List BWT ==
\texttt{stoneInvertible?}_reg(p,ts)$regsetgcdpack

if (R has RetractableTo(Integer)) and (V has ConvertibleTo(Symbol)) then

if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
if not empty? s1 then _
\hspace{1cm} output(s1, (convert(p)@String)::OutputForm)$OutputPackage
output(" ")$OutputPackage
if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
empty? s2 \Rightarrow \text{void()}
\hspace{1cm} output(s2, (" ")::OutputForm)$OutputPackage
lp: List P := members(ts)
for q in lp repeat
\hspace{1cm} output((convert(q)@String)::OutputForm)$OutputPackage
output(" ")$OutputPackage
else

if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
output(" ")$OutputPackage
if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
output(" ")$OutputPackage

-- ASSUME p is invertible w.r.t. ts
-- ASSUME mvar(p) is algebraic w.r.t. ts
\hspace{1cm} v := mvar(p)
\hspace{1cm} ts_v := select(ts,v)::P
\hspace{1cm} if mdeg(p) < mdeg(ts_v)
\hspace{1cm} then
\hspace{1cm} \hspace{1cm} hesrg: Record (gcd : P, coef2 : P) := _
\hspace{1cm} \hspace{1cm} \hspace{1cm} halfExtendedSubResultantGcd2(ts_v,p)$P
normalizedAssociate(p:P,ts:TS): P ==
-- ASSUME p is invertible or zero w.r.t. ts
empty?(ts) => p
zero?(p) => p
ground?(p) => 1
zero? initiallyReduce(init(p),ts) =>
  error "in normalizedAssociate$NORMPK: bad #1"
vp := mvar(p)
ip: P := p
mp: P := 1
tp: P := 0
while not ground?(ip) repeat
  v := mvar(ip)
  if algebraic?(v,ts)
    then
      if v = vp
        then
          ts_v := select(ts,v):P
          ip := lastSubResultant(ip,ts_v)$P
          ip := remainder(ip,ts).polnum
          -- ip := primitivePart stronglyReduce(ip,ts)
          ip := primitivePart initiallyReduce(ip,ts)
        else
          qr := recip(ip,ts)
          ip := qr.den
          tp := qr.num * tp
          zero? ip =>
            outputArgs("p = ", " ts = ",p,ts)
            error _
              "in normalizedAssociate$NORMPK: should never happen !"
        else
          tp := tail(ip) * mp + tp
          mp := mainMonomial(ip) * mp
          ip := init(ip)
\[
\begin{align*}
   r & := ip \times mp + tp \\
   r & := \text{remainder}(r, ts).\text{polnum} \\
   \text{-- primitivePart stronglyReduce}(r, ts) \\
   \text{primitivePart initiallyReduce}(r, ts)
\end{align*}
\]

\[
\begin{align*}
\text{normalize}(p: P, ts: TS): \text{List PWT} & = \\
\text{zero? } p & \Rightarrow [[p, ts]@PWT] \\
\text{ground? } p & \Rightarrow [[1, ts]@PWT] \\
\text{zero? initiallyReduce}(\text{init}(p), ts) & \Rightarrow \\
   \text{error "in normalize$\text{NORMPK}: \text{init}$(#1) reduces to 0 w.r.t. #2"}
\end{align*}
\]

\[
\begin{align*}
   \text{-- output("Entering normalize")$\text{OutputPackage}} \\
   \text{-- outputArgs("p = ", ts = ",p,ts}) \\
   \text{-- output("Calling normInvertible?")$\text{OutputPackage}} \\
   \text{lbwt: List BWT := normInvertible}(p, ts) \\
   \text{-- output("Result is: ")$\text{OutputPackage}} \\
   \text{-- output(lbwt::OutputForm)$\text{OutputPackage}} \\
   \text{lpwt: List PWT := []} \\
   \text{for bwt in lbwt repeat} \\
   \text{us := bwt.tower} \\
   q := \text{remainder}(p, us).\text{polnum} \\
   q := \text{removeZero}(q, us) \\
   \text{bwt.val} & \Rightarrow \\
   \text{-- output("Calling normalizedAssociate")$\text{OutputPackage}} \\
   \text{-- outputArgs("q = ", us = ",q,us)} \\
   \text{lpwt := cons([normalizedAssociate}(q, us)@P, us]@PWT, lpwt) \\
   \text{-- output("Leaving normalizedAssociate")$\text{OutputPackage}} \\
   \text{zero? q} & \Rightarrow \text{lpwt := cons(($P, us]@PWT, lpwt) \\
   lpwt := \text{concat}(\text{normalize}(q, us)@(List PWT), lpwt)} \\
   \text{lpwt}
\end{align*}
\]

—— NORMPK.dotabb ——

"NORMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"NORMPK" -> "SFRTCAT"

——

package NORMMA NormInMonogenicAlgebra

—— NormInMonogenicAlgebra.input ——

\set break resume
This package implements the norm of a polynomial with coefficients in a monogenic algebra (using resultants).

See Also:
- )show NormInMonogenicAlgebra

Exports:
norm


--- package NORMMA NormInMonogenicAlgebra ---

)abbrev package NORMMA NormInMonogenicAlgebra
++ Author: Manuel Bronstein
++ Date Created: 23 February 1995
++ Date Last Updated: 23 February 1995
++ Description:
++ This package implements the norm of a polynomial with coefficients
++ in a monogenic algebra (using resultants)

NormInMonogenicAlgebra(R, PolR, E, PolE): Exports == Implementation where
  R: GcdDomain
  PolR: UnivariatePolynomialCategory R
  E: MonogenicAlgebra(R, PolR)
  PolE: UnivariatePolynomialCategory E
  SUP ==> SparseUnivariatePolynomial
  Exports ==> with
      norm: PolE -> PolR
      ++ norm q returns the norm of q,
      ++ i.e. the product of all the conjugates of q.
  Implementation ==> add
    import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, SUP PolR)

PolR2SUP: PolR -> SUP PolR
PolR2SUP q == map(x +-> x::PolR, q)

defpol := PolR2SUP(definingPolynomial()$E)
norm q ==
  p:SUP PolR := 0
  while q ~= 0 repeat
    p := p + monomial(1,degree q)$PolR * PolR2SUP lift leadingCoefficient q
    q := reductum q
  primitivePart resultant(p, defpol)

--- NORMMA.dotabb ---

"NORMMA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMMA"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"NORMMA" -> "MONOGEN"
package NORMRETR NormRetractPackage

— NormRetractPackage.input —

)set break resume
)sys rm -f NormRetractPackage.output
)spool NormRetractPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NormRetractPackage
--E 1

)spool
)lisp (bye)

— NormRetractPackage.help —

===============================================
NormRetractPackage examples
===============================================

This package has no description

See Also:
 o )show NormRetractPackage
NormRetractPackage (NORMRETR)

Exports:
Frobenius normFactors retractIfCan

--- package NORMRETR NormRetractPackage ---

)abbrev package NORMRETR NormRetractPackage
++ Description:
++ This package has no description

NormRetractPackage(F, ExtF, SUEx, ExtP, n):C == T where
F : FiniteFieldCategory
ExtF : FiniteAlgebraicExtensionField(F)
SUEx : UnivariatePolynomialCategory ExtF
ExtP : UnivariatePolynomialCategory SUEx
n : PositiveInteger
SUP ==> SparseUnivariatePolynomial
R ==> SUP F
P ==> SUP R

C ==> with
  normFactors : ExtP -> List ExtP
    ++ normFactors(x) undocumented
  retractIfCan : ExtP -> Union(P, "failed")
    ++ retractIfCan(x) undocumented
  Frobenius : ExtP -> ExtP
    ++ Frobenius(x) undocumented

T ==> add
  normFactors(p:ExtP):List ExtP ==
    facs : List ExtP := [p]
    for i in 1..n-1 repeat
      member?(p := Frobenius p), facs) => return facs
      facs := cons(p, facs)
    facs
CHAPTER 15. CHAPTER N

Frobenius(ff:ExtP):ExtP ==
  fft:ExtP:=0
  while ff ^= 0 repeat
    fft:=fft + monomial(map(Frobenius, leadingCoefficient ff),
                        degree ff)
    ff:=reductum ff
  fft

retractIfCan(ff:ExtP):Union(P, "failed") ==
  fft:P:=0
  while ff ^= 0 repeat
    lc : SUEx := leadingCoefficient ff
    plc: SUP F := 0
    while lc ^= 0 repeat
      lclc:ExtF := leadingCoefficient lc
      (retlc := retractIfCan lclc) case "failed" => return "failed"
      plc := plc + monomial(retlc::F, degree lc)
      lc := reductum lc
    fft:=fft+monomial(plc, degree ff)
  ff:=reductum ff
  fft

package NPCOEF NPCoef

— NPCoef.input —
Package for the determination of the coefficients in the lifting process. Used by MultivariateLifting. This package will work for every euclidean domain R which has property F, i.e. there exists a factor operation in R[x].

See Also:
- `show NPCoef`
\texttt{NPCoef(BP,E,OV,R,P) : C == T where}

\begin{verbatim}
OV : OrderedSet
E : OrderedAbelianMonoidSup
R : EuclideanDomain -- with property "F"
BP : UnivariatePolynomialCategory R
P : PolynomialCategory(R,E,OV)

Z ==> Integer
NNI ==> NonNegativeInteger
USP ==> SparseUnivariatePolynomial(P)
Term ==> Record(expt:NNI,pcoef:P)
Detc ==> Record(valexp:NNI,valcoef:P,posit:NNI)
VTerm ==> List(Term)
DetCoef ==> Record(deter:List(USP),dterm:List(VTerm),
                   nfacts:List(BP),nlead:List(P))
TermC ==> Record(coefu:P,detfacts:List(VTerm))
TCoef ==> List(TermC)

C == with
   npcoef : (USP,List(BP),List(P)) -> DetCoef
   listexp : BP -> List(NNI)

T == add
    ---- Local Functions ----
    check : (TermC,Vector P) -> Union(Detc,"failed")
    buildvect : (List(VTerm),NNI) -> Vector(List(VTerm))
    buildtable : (Vector(P),List(List NNI),List P) -> TCoef
    modify : (TCoef,Detc) -> TCoef
    constructp : VTerm -> USP

npcoef(u:USP,factlist:List(BP),leadlist:List(P)) :DetCoef ==
   detcoef:=List(VTerm):=empty();detufact:=List(USP):=empty()
   lexp:=List(List(NNI)):[listexp(v) for v in factlist]
   ulist :Vector(P):=vector [coefficient(u,i) for i in 0..degree u]
   tablecoef:=buildtable(ulist,lexp,leadlist)
   detcoef:=[[ep.first,lcu]$Term for ep in lexp for lcu in leadlist]
   ldtcf:=detcoef
   lexp:=[ep.rest for ep in lexp]
   ndet:NNI:=#factlist
\end{verbatim}
changed:Boolean:=true
ltochange:List(NNI):=empty()
ltdel:List(NNI):=empty()
while changed and ndet=1 repeat
  changed:=false
dt:=#tablecoef
  for i in 1..dt while ~changed repeat
    (cf:=check(tablecoef,i,ulist)) case "failed" => "next i"
    ltochange:=cons(i,ltochange)
    celf:Detc:=cf::Detc
    tablecoef:=modify(tablecoef,celf)
    vpos:=celf.posit
    vexp:=celf.valexp
    nterm:=[vexp,celf.valcoef]$Term
    detcoef.vpos:=cons(nterm,detcoef.vpos)
    lexp.vpos:=delete(lexp.vpos,position(vexp,lexp.vpos))
    if lexp.vpos=[] then
      ltdel:=cons(vpos,ltodel)
      ndet:=(ndet-1):NNI
      detufact:=cons(constructp(detcoef.vpos),detufact)
      changed:=true
    end if
  end for
  for i in ltochange repeat tablecoef:=delete(tablecoef,i)
  ltochange:=[ ]
if ndet=1 then
  uu:=u exquo */[pol for pol in detufact]
  if uu case "failed" then return
  [empty(),ldtcf,factlist,leadlist]$DetCoef
else
  detufact:=cons(uu::USP,detufact)
else
  ltodel:=sort((n1:NNI,n2:NNI):Boolean +-> n1>n2,ltodel)
  for i in ltodel repeat
    detcoef:=delete(detcoef,i)
    factlist:=delete(factlist,i)
    leadlist:=delete(leadlist,i)
  end for
[detufact,detcoef,factlist,leadlist]$DetCoef

check(tterm:TermC,ulist:Vector(P)) : Union(Detc,"failed") ==
cfu:P:=1$P;doit:NNI:=0;poselt:NNI:=0;pp:Union(P,"failed")
termlist:List(VTerm):=tterm.detfacts
vterm:VTerm:=$empty()
#termlist=1 =>
  vterm:=termlist.first
  for elterm in vterm while doit<2 repeat
    (cu1:=elterm.pcoef)^=0 => cfu:=cu1*cfu
doit:=doit+1
    poselt:=position(elterm,vterm):NNI
    doit=2 or (pp:=tterm.coefu exquo cfu) case "failed" => "failed"
    [vterm.poselt.expt,pp::P,poselt]$Detc
"failed"
buildvect(lvterm:List(VTerm),n:NNI) : Vector(List(VTerm)) ==
    vtable:Vector(List(VTerm)):=new(n,empty())
    (#lvterm)=1 =>
        for term in lvterm.first repeat vtable.(term.expt+1):=[term]
    vtable
    vtable:=buildvect(lvterm.rest,n)
    ntable:Vector(Vector(List(VTerm))):=new(n,empty())
    for term in lvterm.first repeat
        nexp:=term.expt
        for i in 1..n while (nexp+i)<(n+1) repeat
            ntable.(nexp+i):=append(
                [cons(term,lvterm) for lvterm in vtable.i],
                ntable.(nexp+i))
    ntable

buildtable(vu:Vector(P),lvect:List(List(NNI)),leadlist:List(P)):TCoef==
    nfact:NNI:=#leadlist
    table:TCoef:=empty()
    degu:=(#vu-1)::NNI
    prelim:List(VTerm):=[[[e,0]$P$Term for e in lv] for lv in lvect]
    for i in 1..nfact repeat prelim.i.first.pcoef:=leadlist.i
    partialv:Vector(List(VTerm)):=new(nfact,empty())
    partialv:=buildvect(prelim,degu)
    for i in 1..degu repeat
        empty? partialv.i => "next i"
    table:=cons([vu.i,partialv.i]$TermC, table)
    table

modify(tablecoef:TCoef,cfter:Detc) : TCoef ==
    cfexp:=cfter.valexp; cfcoef:=cfter.valcoef; cfpos:=cfter.posit
    lterase:List(NNI):=empty()
    for cterm in tablecoef | "empty?(ctdet:=cterm.detfacts) repeat
        ([/term.expt for term in ctdet.first])<cfexp => "next term"
        for celt in ctdet repeat
            if celt.cfpos.expt=cfexp then
                celt.cfpos.pcoef:=cfcoef
            if (and/[cc.pcoef ^=0 for cc in celt]) then
                k:=position(ctelt,ctdet):NNI
                lterase:=cons(k,lterase)
                cterm.coefu:=(cterm.coefu - */[cc.pcoef for cc in celt])
            if not empty? lterase then
                lterase:=sort((n1:NNI,n2:NNI):Boolean +-> n1>n2,lterase)
                for i in lterase repeat ctdet:=delete(ctdet,i)
                cterm.detfacts:=ctdet
                lterase:=empty()
    tablecoef

listexp(up:BP) : List(NNI) ==
degree up=0 => [0]
[degree up,.listexp(reductum up)]

constructp(lterm:VTerm):USP ==
+/[monomial(term.pcoef,term.expt) for term in lterm]

---

— NPCOEF.dotabb —

"NPCOEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPCOEF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"NPCOEF" -> "PFECAT"

---

package NFINTBAS NumberFieldIntegralBasis

— NumberFieldIntegralBasis.input —

)set break resume
)sys rm -f NumberFieldIntegralBasis.output
)spool NumberFieldIntegralBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumberFieldIntegralBasis
--E 1

)spool
)lisp (bye)

---

— NumberFieldIntegralBasis.help —

====================================================================
NumberFieldIntegralBasis examples
====================================================================

In this package F is a framed algebra over the integers (typically
F = Z[a] for some algebraic integer a). The package provides functions to compute the integral closure of Z in the quotient field of F.

See Also:
- )show NumberFieldIntegralBasis

NumberFieldIntegralBasis (NFINTBAS)

Exports:
discriminant integralBasis localIntegralBasis

— package NFINTBAS NumberFieldIntegralBasis —

)abbrev package NFINTBAS NumberFieldIntegralBasis
++ Author: Victor Miller, Clifton Williamson
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Description:
++ In this package F is a framed algebra over the integers (typically \spad{F = Z[a]} for some algebraic integer a). The package provides ++ functions to compute the integral closure of Z in the quotient ++ field of F.

NumberFieldIntegralBasis(UP,F): Exports == Implementation where
   UP : UnivariatePolynomialCategory Integer
   F : FramedAlgebra(Integer,UP)

   FR ==> Factored Integer
   I  ==> Integer
   Mat ==> Matrix I
NNI ==> NonNegativeInteger
Ans ==> Record(basis: Mat, basisDen: I, basisInv: Mat, discr: I)

Exports ==> with
  discriminant: () -> Integer
  ++ \spad{discriminant()} returns the discriminant of the integral
  ++ closure of \( \mathbb{Z} \) in the quotient field of the framed algebra \( F \).
  integralBasis: () -> Record(basis: Mat, basisDen: I, basisInv: Mat)
  ++ \spad{integralBasis()} returns a record
  ++ \spad{\{basis, basisDen, basisInv\}}
  ++ containing information regarding the integral closure of \( \mathbb{Z} \) in the
  ++ quotient field of \( F \), where \( F \) is a framed algebra with \( \mathbb{Z} \)-module
  ++ basis \( \spad{\{w_1, w_2, \ldots, w_n\}} \).
  ++ If \( \spad{basis} \) is the matrix \( \spad{\{aij, i = 1..n, j = 1..n\}} \), then
  ++ the \( i \)th element of the integral basis is
  ++ \( \spad{\{vi = (1/basisDen) * sum(aij * w_j, j = 1..n)\}} \), i.e. the
  ++ \( i \)th row of \( \spad{basis} \) contains the coordinates of the
  ++ \( i \)th basis vector. Similarly, the \( i \)th row of the
  ++ matrix \( \spad{basisInv} \) contains the coordinates of \( \spad{wi} \) with
  ++ respect to the basis \( \spad{\{v_1, \ldots, v_n\}} \): if \( \spad{basisInv} \) is the
  ++ matrix \( \spad{\{bij, i = 1..n, j = 1..n\}} \), then
  ++ \( \spad{vi = sum(bij * v_j, j = 1..n)} \).
  localIntegralBasis: I -> Record(basis: Mat, basisDen: I, basisInv: Mat)
  ++ \spad{integralBasis(p)} returns a record
  ++ \spad{\{basis, basisDen, basisInv\}} containing information regarding
  ++ the local integral closure of \( \mathbb{Z} \) at the prime \( \spad{p} \) in the quotient
  ++ field of \( F \), where \( F \) is a framed algebra with \( \mathbb{Z} \)-module basis
  ++ \( \spad{\{w_1, w_2, \ldots, w_n\}} \).
  ++ If \( \spad{basis} \) is the matrix \( \spad{\{aij, i = 1..n, j = 1..n\}} \), then
  ++ the \( i \)th element of the integral basis is
  ++ \( \spad{\{vi = (1/basisDen) * sum(aij * w_j, j = 1..n)\}} \), i.e. the
  ++ \( i \)th row of \( \spad{basis} \) contains the coordinates of the
  ++ \( i \)th basis vector. Similarly, the \( i \)th row of the
  ++ matrix \( \spad{basisInv} \) contains the coordinates of \( \spad{wi} \) with
  ++ respect to the basis \( \spad{\{v_1, \ldots, v_n\}} \): if \( \spad{basisInv} \) is the
  ++ matrix \( \spad{\{bij, i = 1..n, j = 1..n\}} \), then
  ++ \( \spad{wi = sum(bij * v_j, j = 1..n)} \).

Implementation ==> add
  import IntegralBasisTools(I, UP, F)
  import ModularHermitianRowReduction(I)
  import TriangularMatrixOperations(I, Vector I, Vector I, Matrix I)

frobMatrix : (Mat, Mat, I, NNI) -> Mat
wildPrimes : (FR, I) -> List I
tameProduct : (FR, I) -> I
iTameLocalIntegralBasis : (Mat, I, I) -> Ans
iWildLocalIntegralBasis : (Mat, I, I) -> Ans

frobMatrix(rb, rbinv, rbden, p) ==
n := rank(F); b := basis(F)
v : Vector F := new(n,0) for i in minIndex(v)..maxIndex(v) repeat
  a : F := 0 for j in minIndex(b)..maxIndex(b) repeat
    for jj in minColIndex(rb)..maxColIndex(rb) repeat
      a := a + qelt(rb,ii,jj) * qelt(b,j)
    qsetelt!(v,i,a**p)
mat := transpose coordinates v ((transpose(rbinv) * mat) exquo (rbden ** p)) :: Mat

wildPrimes(factoredDisc,n) ==
  -- returns a list of the primes <=n which divide factoredDisc to a
  -- power greater than 1
  ans : List I := empty() for f in factors(factoredDisc) repeat
    if f.exponent > 1 and f.factor <= n then ans := concat(f.factor,ans)
  ans

tameProduct(factoredDisc,n) ==
  -- returns the product of the primes > n which divide factoredDisc
  -- to a power greater than 1
  ans : I := 1 for f in factors(factoredDisc) repeat
    if f.exponent > 1 and f.factor > n then ans := f.factor * ans
  ans

integralBasis() ==
  traceMat := traceMatrix(F); n := rank(F)
disc := determinant traceMat -- discriminant of current order
disc0 := disc -- this is disc(F)
factoredDisc := factor(disc0)$IntegerFactorizationPackage
wilds := wildPrimes(factoredDisc,n)
sing := tameProduct(factoredDisc,n)
runningRb := scalarMatrix(n, 1); runningRbinv := scalarMatrix(n, 1)
-- runningRb = basis matrix of current order
-- runningRbinv = inverse basis matrix of current order
-- these are wrt the original basis for F
runningRbden : I := 1
-- runningRbden = denominator for current basis matrix
  (sing = 1) and empty? wilds => [runningRb, runningRbden, runningRbinv] for p in wilds repeat
    lb := iWildLocalIntegralBasis(matrixOut,disc,p)
    rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
disc := lb.dscr
    -- update 'running integral basis' if newly computed
-- local integral basis is non-trivial
if sizeLess?(1,rbden) then
  mat := vertConcat(rbdn * runningRb, runningRbden * rb)
  runningRbden := runningRbden * rbden
  runningRb := squareTop rowEchelon(mat, runningRbden)
  runningRbinv := UpTriBddDenomInv(runningRb, runningRbden)
  lb := iTameLocalIntegralBasis(traceMat, disc, sing)
  rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
  disc := lb/disc
  -- update 'running integral basis' if newly computed
  -- local integral basis is non-trivial
  if sizeLess?(1,rbden) then
    mat := vertConcat(rbdn * runningRb, runningRbden * rb)
    runningRbden := runningRbden * rbden
    runningRb := squareTop rowEchelon(mat, runningRbden)
    runningRbinv := UpTriBddDenomInv(runningRb, runningRbden)
    [runningRb, runningRbden, runningRbinv]

localIntegralBasis p ==
  traceMat := traceMatrix()$F; n := rank()$F
  disc := determinant traceMat -- discriminant of current order
  (disc exquo (p*p)) case "failed" =>
    [scalarMatrix(n, 1), 1, scalarMatrix(n, 1)]
  lb :=
    p > rank()$F =>
      iTameLocalIntegralBasis(traceMat, disc, p)
      iWildLocalIntegralBasis(scalarMatrix(n,0), disc, p)
    [lb.basis, lb.basisDen, lb.basisInv]

iTameLocalIntegralBasis(traceMat, disc, sing) ==
  n := rank()$F; disc0 := disc
  rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
  -- rb = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  rbden : I := 1; index : I := 1; oldIndex : I := 1
  -- rbden = denominator for current basis matrix
  -- id = basis matrix of the ideal (p-radical) wrt current basis
  tfm := traceMat
  repeat
    -- compute the p-radical = p-trace-radical
    idinv := transpose squareTop rowEchelon(tfm, sing)
    -- [u1,...,un] are the coordinates of an element of the p-radical
    -- iff [u1,...,un] * idinv is in p * Z^n
    id := rowEchelon LowTriBddDenomInv(idinv, sing)
    -- id = basis matrix of the p-radical
    idinv := UpTriBddDenomInv(id, sing)
    -- id * idinv = sing * identity
    -- no need to check for inseparability in this case
    rbinv := idealiser(id * rb, rbinv * idinv, sing * rbden)

index := diagonalProduct rbinv
rb := rowEchelon LowTriBddDenomInv(rbinv, sing * rbden)
g := matrixGcd(rb, sing, n)
if sizeLess?(1, g) then rb := (rb exquo g) :: Mat
rbden := rbden * (sing quo g)
rbinv := UpTriBddDenomInv(rb, rbden)
disc := disc0 quo (index * index)

indexChange := index quo oldIndex; oldIndex := index
-- one? indexChange => return [rb, rbden, rbinv, disc]
(indexChange = 1) => return [rb, rbden, rbinv, disc]
tfm := ((rb * traceMat * transpose rb) exquo (rbden * rbden)) :: Mat

iWildLocalIntegralBasis(matrixOut, disc, p) ==
    n := rank()$F; disc0 := disc
    rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
    -- rb = basis matrix of current order
    -- rbinv = inverse basis matrix of current order
    -- these are wrt the original basis for F
    rbden := I := 1; index := I := 1; oldIndex := I := 1
    -- rbden = denominator for current basis matrix
    -- id = basis matrix of the ideal (p-radical) wrt current basis
    p2 := p * p; lp := leastPower(p::NNI, n)
    repeat
        tfm := frobMatrix(rb, rbinv, rbden, p::NNI) ** lp
        -- compute Rp = p-radical
        idinv := transpose squareTop rowEchelon(tfm, p)
        -- [u1, ..., un] are the coordinates of an element of Rp
        -- if [u1, ..., un] * idinv is in p * Z^n
        id := rowEchelon LowTriBddDenomInv(idinv, p)
        -- id = basis matrix of the p-radical
        idinv := UpTriBddDenomInv(id, p)
        -- id * idinv = p * identity
        -- no need to check for inseparability in this case
        rbinv := idealiser(id * rb, rbinv * idinv, p * rbden)
        index := diagonalProduct rbinv
        rb := rowEchelon LowTriBddDenomInv(rbinv, p * rbden)
        if divideIfCan!(rb, matrixOut, p, n) = 1
            then rb := matrixOut
        else rbden := p * rbden
        rbinv := UpTriBddDenomInv(rb, rbden)
        indexChange := index quo oldIndex; oldIndex := index
        disc := disc quo (indexChange * indexChange)
        -- one? indexChange or gcd(p2, disc) ^= p2 =>
        (indexChange = 1) or gcd(p2, disc) ^= p2 =>
            return [rb, rbden, rbinv, disc]
        discriminant() ==
            disc := determinant traceMatrix()$F
            intBas := integralBasis()
            rb := intBas.basis; rbden := intBas.basisDen
index := ((rbden ** rank()$F) exquo (determinant rb)) :: Integer
(disc exquo (index * index)) :: Integer

---

package NUMFMT NumberFormats

--- NumberFormats.input ---

)set break resume
)sys rm -f NumberFormats.output
)spool NumberFormats.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumberFormats
--E 1

)spool
)lisp (bye)

---

--- NumberFormats.help ---

====================================================================
NumberFormats examples
====================================================================

NumberFormats provides function to format and read arabic and roman numbers, to convert numbers to strings and to read floating-point numbers.

See Also:
NumbersFormats (NUMFMT)

Exports:
FormatArabic FormatRoman ScanArabic ScanFloatIgnoreSpaces
ScanFloatIgnoreSpacesIfCan ScanRoman

— package NUMFMT NumberFormats —

)abbrev package NUMFMT NumberFormats
++ SMW March 88
++ Description:
++ NumbersFormats provides function to format and read arabic and
++ roman numbers, to convert numbers to strings and to read
++ floating-point numbers.

NumbersFormats(): NFexports == NFimplementation where
PI ==> PositiveInteger
I ==> Integer
C ==> Character
F ==> Float
S ==> String
V ==> PrimitiveArray

NFexports ==> with
FormatArabic: PI -> S
   ++ FormatArabic(n) forms an Arabic numeral
   ++ string from an integer n.
ScanArabic: S -> PI
   ++ ScanArabic(s) forms an integer from an Arabic numeral string s.
FormatRoman: PI -> S
++ FormatRoman(n) forms a Roman numeral string from an integer n.
ScanRoman: S -> PI
++ ScanRoman(s) forms an integer from a Roman numeral string s.
ScanFloatIgnoreSpaces: S -> F
++ ScanFloatIgnoreSpaces(s) forms a floating point number from
++ the string s ignoring any spaces. Error is generated if the
++ string is not recognised as a floating point number.
ScanFloatIgnoreSpacesIfCan: S -> Union(F, "failed")
++ ScanFloatIgnoreSpacesIfCan(s) tries to form a floating point
++ number from the string s ignoring any spaces.

NFimplementation ==> add
import SExpression
import Symbol
replaceD: C -> C
replaced: C -> C
contract: S -> S
check: S ->Boolean
replaceD c ==
  if c = char "D" then char "E" else c
replaced c ==
  if c = char "d" then char "E" else c
contract s ==
s:= map(replaceD,s)
s:= map(replaced,s)
ls:List S := split(s,char " ")$String
s:= concat ls
check s ==
  NUMBERP(READ_FROM_STRING(s)$Lisp)$Lisp and
  -- if there is an "E" then there must be a "."
  -- this is not caught by code above
  -- also if the exponent is v.big the above returns false
  not (any?((c1:C):Boolean +-> c1=char "E",s)
    and not any?((c2:C):Boolean +-> c2=char ".",s) )

-- Original interpreter function:
-- )lis (defun scanstr(x) (spadcomp:|parseFromString| x))
sexfloat:SExpression:=convert(coerce("Float")@Symbol)$SExpression
ScanFloatIgnoreSpaces s ==
s := contract s
not check s => error "Non-numeric value"
sex := interpret(ncParseFromString(s)$Lisp)$Lisp
sCheck := car(car(sex))
if (sCheck=sexfloat) = true then
  f := (cdr cdr sex) pretend Float
else
  if integer?(cdr sex) = true then
    f := (cdr sex) pretend Integer
  f::F
else
    error "Non-numeric value"

ScanFloatIgnoreSpacesIfCan s ==
    s := contract s
    not check s => "failed"
    sex := interpret(ncParseFromString(s)$Lisp)$Lisp
    sCheck := car(car(sex))
    if (sCheck=sexfloat) = true then
        f := (cdr cdr sex) pretend Float
    else
        if integer?(cdr sex) = true then
            f := (cdr sex) pretend Integer
            f::F
        else
            "failed"
    else
        "failed"

units:V S :=
    construct ['"","I","II","III","IV","V","VI","VII","VIII","IX"]
tens :V S :=
    construct ['"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"]
hunds:V S :=
    construct ['"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"]
umin := minIndex units
tmin := minIndex tens
hmin := minIndex hunds
romval:V I := new(256, -1)
romval ord char(" ")$C := 0
romval ord char("I")$C := 1
romval ord char("V")$C := 5
romval ord char("X")$C := 10
romval ord char("L")$C := 50
romval ord char("C")$C := 100
romval ord char("D")$C := 500
romval ord char("M")$C := 1000
thou:C := char "M"
plen:C := char "("
pren:C := char ")"
ichar:C := char "I"

FormatArabic n == STRINGIMAGE(n)$Lisp
ScanArabic s == PARSE_-INTEGER(s)$Lisp

FormatRoman pn ==
    n := pn::Integer
    -- Units
    d := (n rem 10) + umin
    n := n quo 10
    s := units.d
    zero? n => s
-- Tens
d := (n rem 10) + tmin
n := n quo 10
s := concat(tens.d, s)
zero? n => s
-- Hundreds
d := (n rem 10) + hmin
n := n quo 10
s := concat(hunds.d, s)
zero? n => s
-- Thousands
d := n rem 10
n := n quo 10
s := concat(new(d::NonNegativeInteger, thou), s)
zero? n => s
-- Ten thousand and higher
for i in 2.. while not zero? n repeat
   -- Coefficient of 10**(i+2)
   d := n rem 10
   n := n quo 10
   zero? d => "iterate"
   m0:String := concat(new(i,plen),concat("I",new(i,pren)))
   mm := concat([m0 for j in 1..d]$List(String))
   if #s > 0 then s := concat(" ", s)
   s := concat(mm, s)
s
-- ScanRoman
--
-- The Algorithm:
-- Read number from right to left. When the current
-- numeral is lower in magnitude than the previous maximum
-- then subtract otherwise add.
-- Shift left and repeat until done.
ScanRoman s ==
s := upperCase s
tot: I := 0
Max: I := 0
i: I := maxIndex s
while i >= minIndex s repeat
   -- Read a single roman digit
   c := s.i; i := i-1
   n := romval ord c
   -- (I)=1000, ((I))=10000, (((I)))=100000, etc
   if n < 0 then
      c ^= pren =>
      error ["Improper character in Roman numeral: ",c]
nprens: PI := 1
while c = pren and i >= minIndex s repeat
    c := s.i; i := i-1
    if c = pren then nprens := nprens+1
    c ^= ichar =>
        error "Improper Roman numeral: (x)"
for k in 1..nprens while i >= minIndex s repeat
    c := s.i; i := i-1
    c ^= plen =>
        error "Improper Roman numeral: unbalanced ')'"
    n := 10**(nprens + 2)
    if n < Max then
        tot := tot - n
    else
        tot := tot + n
        Max := n
    tot < 0 => error ["Improper Roman numeral: ", tot]
tot::PI

— NUMFMT.dotabb —

"NUMFMT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMFMT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NUMFMT" -> "ALIST"

——

package NTPOLFN NumberTheoreticPolynomialFunctions

— NumberTheoreticPolynomialFunctions.input —

)set break resume
)sys rm -f NumberTheoreticPolynomialFunctions.output
)spool NumberTheoreticPolynomialFunctions.output
)set message test on
)set message auto off
)cold all

--S 1 of 1
)show NumberTheoreticPolynomialFunctions
--E 1

)spool
)lisp (bye)
NumberTheoreticPolynomialFunctions

This package provides polynomials as functions on a ring.

See Also:
o  )show NumberTheoreticPolynomialFunctions

Exports:
  bernoulliB  cyclotomic  eulerE

-- package NTPOLFN NumberTheoreticPolynomialFunctions --

)abbrev package NTPOLFN NumberTheoreticPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created:  1990
++ Date Last Updated: June 25, 1991
++ Description:
++ This package provides polynomials as functions on a ring.

NumberTheoreticPolynomialFunctions(R: CommutativeRing): Exports == Impl where
  NNI ==> NonNegativeInteger
  RN  ==> Fraction Integer
Exports => with

    cyclotomic: (NNI, R) -> R
    ++ cyclotomic(n,r) \undocumented

if R has Algebra RN then
    bernoulliB: (NNI, R) -> R
    ++ bernoulliB(n,r) \undocumented
eulerE: (NNI, R) -> R
    ++ eulerE(n,r) \undocumented

Impl => add

import PolynomialNumberTheoryFunctions()

I => Integer
SUP => SparseUnivariatePolynomial

-- This is the wrong way to evaluate the polynomial.
cyclotomic(k, x) ==
p: SUP(I) := cyclotomic(k)
r: R := 0
while p ^= 0 repeat
    d := degree p
c := leadingCoefficient p
p := reductum p
r := c*x**d + r
r

if R has Algebra RN then
eulerE(k, x) ==
p: SUP(RN) := euler(k)
r: R := 0
while p ^= 0 repeat
    d := degree p
c := leadingCoefficient p
p := reductum p
r := c*x**d + r
r

bernoulliB(k, x) ==
p: SUP(RN) := bernoulli(k)
r: R := 0
while p ^= 0 repeat
    d := degree p
c := leadingCoefficient p
p := reductum p
r := c*x**d + r
r
package NUMERIC Numeric

---

Numeric examples

---

Numeric provides real and complex numerical evaluation functions for various symbolic types.

See Also:
- )show Numeric
Numeric (NUMERIC)

Exports:
numeric complexNumeric complexNumericIfCan numeric numericIfCan

— package NUMERIC Numeric —

)abbrev package NUMERIC Numeric
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 24 January 1997
++ Description:
++ Numeric provides real and complex numerical evaluation
++ functions for various symbolic types.

Numeric(S:ConvertibleTo Float): with
numeric: S -> Float
++ numeric(x) returns a real approximation of x.
numeric: (S, PositiveInteger) -> Float
++ numeric(x, n) returns a real approximation of x up to n decimal
++ places.
complexNumeric: S -> Complex Float
++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (S, PositiveInteger) -> Complex Float
++ complexNumeric(x, n) returns a complex approximation of x up
++ to n decimal places.
if S has CommutativeRing then
complexNumeric: Complex S -> Complex Float
++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Complex S, PositiveInteger) -> Complex Float
++ complexNumeric(x, n) returns a complex approximation of x up
++ to n decimal places.
complexNumeric: Polynomial Complex S -> Complex Float
++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Polynomial Complex S, PositiveInteger) -> Complex Float
++ complexNumeric(x, n) returns a complex approximation of x up
++ to n decimal places.
if S has Ring then
  numeric: Polynomial S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (Polynomial S, PositiveInteger) -> Float
    ++ numeric(x, n) returns a real approximation of x up to n decimal
    ++ places.
complexNumeric: Polynomial S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Polynomial S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x
  ++ up to n decimal places.
if S has IntegralDomain then
  numeric: Fraction Polynomial S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (Fraction Polynomial S, PositiveInteger) -> Float
    ++ numeric(x, n) returns a real approximation of x up to n decimal
    ++ places.
complexNumeric: Fraction Polynomial S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Fraction Polynomial S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x
complexNumeric: Fraction Polynomial Complex S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Fraction Polynomial Complex S, PositiveInteger) ->
  Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x
    ++ up to n decimal places.
if S has OrderedSet then
  numeric: Expression S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (Expression S, PositiveInteger) -> Float
    ++ numeric(x, n) returns a real approximation of x up to n
    ++ decimal places.
complexNumeric: Expression S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Expression S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x
  ++ up to n decimal places.
complexNumeric: Expression Complex S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Expression Complex S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x
  ++ up to n decimal places.
if S has CommutativeRing then
complexNumericIfCan: Polynomial Complex S -> Union(Complex Float, "failed")
  ++ complexNumericIfCan(x) returns a complex approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not constant.
complexNumericIfCan: (Polynomial Complex S, PositiveInteger) -> Union(Complex Float, "failed")
  ++ complexNumericIfCan(x, n) returns a complex approximation of x up
  ++ to n decimal places, or "failed" if \( \text{axiom}(x) \) is not a constant.
if S has Ring then
  numericIfCan: Polynomial S -> Union(Float,"failed")
  ++ numericIfCan(x) returns a real approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  numericIfCan: (Polynomial S, PositiveInteger) -> Union(Float,"failed")
  ++ numericIfCan(x,n) returns a real approximation of x up to n decimal
  ++ places, or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: Polynomial S -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x) returns a complex approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: (Polynomial S, PositiveInteger) -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x, n) returns a complex approximation of x
  ++ up to n decimal places, or "failed" if \( \text{axiom}(x) \) is not a constant.
if S has IntegralDomain then
  numericIfCan: Fraction Polynomial S -> Union(Float,"failed")
  ++ numericIfCan(x) returns a real approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  numericIfCan: (Fraction Polynomial S, PositiveInteger) -> Union(Float,"failed")
  ++ numericIfCan(x,n) returns a real approximation of x up to n decimal
  ++ places, or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: Fraction Polynomial S -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x) returns a complex approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: (Fraction Polynomial S, PositiveInteger) -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x, n) returns a complex approximation of x
  ++ up to n decimal places, or "failed" if \( \text{axiom}(x) \) is not a constant.
if S has OrderedSet then
  numericIfCan: Expression S -> Union(Float,"failed")
  ++ numericIfCan(x) returns a real approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  numericIfCan: (Expression S, PositiveInteger) -> Union(Float,"failed")
  ++ numericIfCan(x,n) returns a real approximation of x up to n
  ++ decimal places, or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: Expression S -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x) returns a complex approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: (Expression S, PositiveInteger) ->
  Union(Complex Float,"failed")
  ++ complexNumericIfCan(x, n) returns a complex approximation of x
  ++ up to n decimal places, or "failed" if \( \text{axiom}(x) \) is not a constant.
  complexNumericIfCan: Expression Complex S -> Union(Complex Float,"failed")
  ++ complexNumericIfCan(x) returns a complex approximation of x,
  ++ or "failed" if \( \text{axiom}(x) \) is not a constant.
complexNumericIfCan: (Expression Complex S, PositiveInteger) -> 
Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x
++ up to n decimal places, or "failed" if \(\text{axiom}(x)\) is not a constant.

== add

if S has CommutativeRing then
complexNumericIfCan(p:Polynomial Complex S) ==
p' : Union(Complex(S),"failed") := retractIfCan p
p' case "failed" => "failed"
complexNumeric(p')

complexNumericIfCan(p:Polynomial Complex S,n:PositiveInteger) ==
p' : Union(Complex(S),"failed") := retractIfCan p
p' case "failed" => "failed"
complexNumeric(p',n)

if S has Ring then
numericIfCan(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" => "failed"
numeric(p')

complexNumericIfCan(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" => "failed"
complexNumeric(p')

complexNumericIfCan(p:Polynomial S, n:PositiveInteger) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" => "failed"
complexNumeric(p', n)

numericIfCan(p:Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numericIfCan p
digits(old)$Float
ans

if S has IntegralDomain then
numericIfCan(f:Fraction Polynomial S)==
num := numericIfCan(numer(f))
num case "failed" => "failed"
den := numericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial S) ==
num := complexNumericIfCan(numer f)
num case "failed" => "failed"
den := complexNumericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
num := complexNumericIfCan(numer f, n)
num case "failed" => "failed"
den := complexNumericIfCan(denom f, n)
den case "failed" => "failed"
num/den

numericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numericIfCan f
digits(old)$Float
ans

complexNumericIfCan(f:Fraction Polynomial Complex S) ==
num := complexNumericIfCan(numer f)
num case "failed" => "failed"
den := complexNumericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
num := complexNumericIfCan(numer f, n)
num case "failed" => "failed"
den := complexNumericIfCan(denom f, n)
den case "failed" => "failed"
num/den

if S has OrderedSet then
numericIfCan(x:Expression S) ==
  retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
--s2cs(u:S):Complex(S) == complex(u,0)

complexNumericIfCan(x:Expression S) ==
  complexNumericIfCan(map(coerce, x)$ExpressionFunctions2(S, Complex S)

numericIfCan(x:Expression S, n:PositiveInteger) ==
old := digits(n)$Float
x' : Expression Float := map(convert, x)$ExpressionFunctions2(S, Float)
anl : Union(Float,"failed") := retractIfCan x'
digits(old)$Float
ans

complexNumericIfCan(x:Expression S, n:PositiveInteger) ==
old := digits(n)$Float
x' : Expression Complex S := map(coerce, x)$ExpressionFunctions2(S, Complex S)
ans : Union(Complex Float,"failed") := complexNumericIfCan(x')
digits(old)$Float
ans

if S has RealConstant then
  complexNumericIfCan(x:Expression Complex S) ==
    retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Complex Float :=
    map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
  ans : Union(Complex Float,"failed") := retractIfCan x'
  digits(old)$Float
  ans
else
  convert(x:Complex S):Complex(Float)==map(convert,x)$ComplexFunctions2(S,Float)

complexNumericIfCan(x:Expression Complex S) ==
  retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Complex Float :=
    map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
  ans : Union(Complex Float,"failed") := retractIfCan x'
  digits(old)$Float
  ans
numeric(s:S) == convert(s)@Float
if S has ConvertibleTo Complex Float then
  complexNumeric(s:S) == convert(s)@Complex(Float)

complexNumeric(s:S, n:PositiveInteger) ==
  old := digits(n)$Float
  ans := complexNumeric s
digits(old)$Float
  ans
else
  complexNumeric(s:S) == convert(s)@Float :: Complex(Float)

complexNumeric(s:S,n:PositiveInteger) ==
  numeric(s, n)::Complex(Float)
if S has CommutativeRing then
  complexNumeric(p:Polynomial Complex S) ==
    p' : Union(Complex(S),"failed") := retractIfCan p
    p' case "failed" =>
      error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p')

complexNumeric(p:Polynomial Complex S,n:PositiveInteger) ==
p' : Union(Complex(S),"failed") := retractIfCan p
p' case "failed" =>
  error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p',n)

if S has RealConstant then
  complexNumeric(s:Complex S) == convert(s)$Complex(S)
complexNumeric(s:Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  ans := complexNumeric s
digits(old)$Float
  ans

else if Complex(S) has ConvertibleTo(Complex Float) then
  complexNumeric(s:Complex S) == convert(s)@Complex(Float)
complexNumeric(s:Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  ans := complexNumeric s
digits(old)$Float
  ans

else
  complexNumeric(s:Complex S) ==
s' : Union(S,"failed") := retractIfCan s
s' case "failed" =>
  error "Cannot compute the numerical value of a non-constant object"
complexNumeric(s')

complexNumeric(s:Complex S, n:PositiveInteger) ==
s' : Union(S,"failed") := retractIfCan s
s' case "failed" =>
  error "Cannot compute the numerical value of a non-constant object"
old := digits(n)$Float
ans := complexNumeric s'
digits(old)$Float
ans

numeric(s:S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric s
digits(old)$Float
ans

if S has Ring then
  numeric(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" =>
   "Can only compute the numerical value of a constant, real-valued polynomial"
   numeric(p')

complexNumeric(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed"
   error "Cannot compute the numerical value of a non-constant polynomial"
   complexNumeric(p')

complexNumeric(p:Polynomial S, n:PositiveInteger) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed"
   error "Cannot compute the numerical value of a non-constant polynomial"
   complexNumeric(p', n)

numeric(p:Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric p
digits(old)$Float
ans

if S has IntegralDomain then
   numeric(f:Fraction Polynomial S)==
   numeric(numer(f))/numeric(denom f)
complexNumeric(f:Fraction Polynomial S) ==
   complexNumeric(numer f)/complexNumeric(denom f)
complexNumeric(f:Fraction Polynomial S, n:PositiveInteger) ==
   complexNumeric(numer f, n)/complexNumeric(denom f, n)

numeric(f:Fraction Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric f
digits(old)$Float
ans

complexNumeric(f:Fraction Polynomial Complex S) ==
   complexNumeric(numer f)/complexNumeric(denom f)
complexNumeric(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
   complexNumeric(numer f, n)/complexNumeric(denom f, n)

if S has OrderedSet then
   numeric(x:Expression S) ==
x' : Union(Float,"failed") :=
   retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
x' case "failed" => error
"Can only compute the numerical value of a constant, real-valued Expression"

\[ x' \]

\[ \text{complexNumeric}(x: \text{Expression } S) == \]
\[ x': \text{Union}(\text{Complex Float}, "failed") := \text{retractIfCan(} \]
\[ \text{map(\text{complexNumeric}, x)$\text{ExpressionFunctions2}(S, \text{Complex Float})} \]
\[ x' \text{ case } "failed" => \]
\[ \text{error "Cannot compute the numerical value of a non-constant expression"} \]
\[ x' \]

\[ \text{numeric}(x: \text{Expression } S, n: \text{PositiveInteger}) == \]
\[ \text{old} := \text{digits(n)$\text{Float} \]
\[ x': \text{Expression Float} := \text{map(convert, x)$\text{ExpressionFunctions2}(S, \text{Float})} \]
\[ \text{ans} : \text{Union}(\text{Float}, "failed") := \text{retractIfCan x'} \]
\[ \text{digits(old)$\text{Float} \]
\[ \text{ans case } "failed" => \text{error} \]
\[ "\text{Can only compute the numerical value of a constant, real-valued Expression"} \]
\[ \text{ans} \]

\[ \text{complexNumeric}(x: \text{Expression } S, n: \text{PositiveInteger}) == \]
\[ \text{old} := \text{digits(n)$\text{Float} \]
\[ x' : \text{Expression Complex Float} := \]
\[ \text{map(\text{complexNumeric}, x)$\text{ExpressionFunctions2}(S, \text{Complex Float})} \]
\[ \text{ans} : \text{Union}(\text{Complex Float}, "failed") := \text{retractIfCan x'} \]
\[ \text{digits(old)$\text{Float} \]
\[ \text{ans case } "failed" => \text{error} \]
\[ "\text{Can only compute the numerical value of a constant, real-valued Expression"} \]
\[ \text{ans} \]

\[ \text{complexNumeric}(x: \text{Expression Complex } S) == \]
\[ x': \text{Union}(\text{Complex Float}, "failed") := \text{retractIfCan(} \]
\[ \text{map(\text{complexNumeric}, x)$\text{ExpressionFunctions2}(\text{Complex } S, \text{Complex Float})} \]
\[ x' \text{ case } "failed" => \]
\[ \text{error "Cannot compute the numerical value of a non-constant expression"} \]
\[ x' \]

\[ \text{complexNumeric}(x: \text{Expression Complex } S, n: \text{PositiveInteger}) == \]
\[ \text{old} := \text{digits(n)$\text{Float} \]
\[ x' : \text{Expression Complex Float} := \]
\[ \text{map(\text{complexNumeric}, x)$\text{ExpressionFunctions2}(\text{Complex } S, \text{Complex Float})} \]
\[ \text{ans} : \text{Union}(\text{Complex Float}, "failed") := \text{retractIfCan x'} \]
\[ \text{digits(old)$\text{Float} \]
\[ \text{ans case } "failed" => \text{error} \]
\[ "\text{Can only compute the numerical value of a non-constant expression"} \]
\[ \text{ans} \]
package NUMODE NumericalOrdinaryDifferentialEquations

— NumericalOrdinaryDifferentialEquations.input —

)set break resume
)sys rm -f NumericalOrdinaryDifferentialEquations.output
)spool NumericalOrdinaryDifferentialEquations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalOrdinaryDifferentialEquations
--E 1

)spool
)lisp (bye)

— NumericalOrdinaryDifferentialEquations.help —

====================================================================
NumericalOrdinaryDifferentialEquations examples
====================================================================

This package is a suite of functions for the numerical integration of an ordinary differential equation of n variables:

\[ \frac{dy}{dx} = f(y,x) \quad y \text{ is an n-vector} \]

All the routines are based on a 4-th order Runge-Kutta kernel.

These routines generally have as arguments:
\( n, \) the number of dependent variables;
\( x1, \) the initial point;
\( h, \) the step size;
\( y, \) a vector of initial conditions of length \( n \)

which upon exit contains the solution at \( x1 + h \)

derivs, a function which computes the right hand side of the ordinary differential equation:

\[
\text{derivs}(\frac{dy}{dx}, y, x)
\]

computes \( \frac{dy}{dx} \), a vector which contains the derivative information.

In order of increasing complexity:

\[
\text{rk4}(y, n, x1, h, \text{derivs})
\]

advances the solution vector to

\( x1 + h \)

and return the values in \( y \).

\[
\text{rk4}(y, n, x1, h, \text{derivs}, t1, t2, t3, t4)
\]

is the same as

\[
\text{rk4}(y, n, x1, h, \text{derivs})
\]

except that you must provide 4 scratch arrays \( t1-t4 \) of size \( n \)

Starting with \( y \) at \( x1 \),

\[
\text{rk4f}(y, n, x1, x2, ns, \text{derivs})
\]

uses \( ns \) fixed steps of a 4-th order Runge-Kutta integrator to advance the solution vector to \( x2 \) and return the values in \( y \). Argument \( x2 \), is the final point, and \( ns \), the number of steps to take.

\[
\text{rk4qc}(y, n, x1, \text{step}, \text{eps}, \text{yscal}, \text{derivs})
\]

takes a 5-th order Runge-Kutta step with monitoring of local truncation to ensure accuracy and adjust stepsize.

The function takes two half steps and one full step and scales the difference in solutions at the final point. If the error is
within eps, the step is taken and the result is returned.

If the error is not within eps, the step size is decreased and the procedure is tried again until the desired accuracy is reached. Upon input, an initial step size must be given and upon return, an estimate of the next step size to use is returned as well as the step size which produced the desired accuracy.

The scaled error is computed as

\[
\text{error} = \text{MAX}(\text{ABS}(\text{y2steps}(i) - \text{y1step}(i))/\text{yscal}(i))
\]

and this is compared against eps. If this is greater than eps, the step size is reduced accordingly to

\[
h_{\text{new}} = 0.9 \times h_{\text{did}} \times (\text{error}/\text{eps})^{(-1/4)}
\]

If the error criterion is satisfied, then we check if the step size was too fine and return a more efficient one. If

\[
\text{error} > \text{eps} \times (6.0E^{-04})
\]

then the next step size should be

\[
h_{\text{next}} = 0.9 \times h_{\text{did}} \times (\text{error}/\text{eps})^{(-1/5)}
\]

Otherwise

\[
h_{\text{next}} = 4.0 \times h_{\text{did}}
\]

is returned.

A more detailed discussion of this and related topics can be found in the book "Numerical Recipies" by W. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, published by Cambridge University Press.

Argument step is a record of 3 floating point numbers (try, did, next), eps is the required accuracy, yscal is the scaling vector for the difference in solutions.

On input, step.try should be the guess at a step size to achieve the accuracy. On output, step.did contains the step size which achieved the accuracy and step.next is the next step size to use.

\[
\text{rk4qc}(y,n,x1,\text{step},\text{eps},\text{yscal},\text{derivs},t1,t2,t3,t4,t5,t6,t7)
\]

is the same as

\[
\text{rk4qc}(y,n,x1,\text{step},\text{eps},\text{yscal},\text{derivs})
\]
except that the user must provide the 7 scratch arrays t1-t7 of size n.

\[ \text{rk4a}(y, n, x1, x2, eps, h, ns, \text{derivs}) \]

is a driver program which uses rk4qc to integrate \( n \) ordinary differential equations starting at \( x1 \) to \( x2 \), keeping the local truncation error to within \( \text{eps} \) by changing the local step size.

The scaling vector is defined as

\[ y\text{scal}(i) = \text{abs}(y(i)) + \text{abs}(h \times \text{dydx}(i)) + \text{tiny} \]

where \( y(i) \) is the solution at location \( x \), \( \text{dydx} \) is the ordinary differential equation’s right hand side, \( h \) is the current step size and tiny is 10 times the smallest positive number representable.

The user must supply an estimate for a trial step size and the maximum number of calls to rk4qc to use. Argument \( x2 \) is the final point, \( \text{eps} \) is local truncation, \( \text{ns} \) is the maximum number of call to rk4qc to use.

See Also:
o )show NumericalOrdinaryDifferentialEquations

---

**NumericalOrdinaryDifferentialEquations (NUMODE)**

Exports:
- rk4
- rk4a
- rk4f
- rk4qc
- rk4qc

---

)abbrev package NUMODE NumericalOrdinaryDifferentialEquations
This package is a suite of functions for the numerical integration of an ordinary differential equation of n variables:
\[
\frac{dy}{dx} = f(y,x)
\] where y is an n-vector. All the routines are based on a 4-th order Runge-Kutta kernel. These routines generally have as arguments:
- n, the number of dependent variables;
- x1, the initial point;
- h, the step size;
- y, a vector of initial conditions of length n;
which upon exit contains the solution at x1 + h;

The function which computes the right hand side of the ordinary differential equation: \texttt{derivs(dydx,y,x)} computes \texttt{dydx}, a vector which contains the derivative information.

In order of increasing complexity:
- \texttt{rk4(y,n,x1,h,derivs)} advances the solution vector to x1 + h and return the values in y.
- \texttt{rk4(y,n,x1,h,derivs,t1,t2,t3,t4)} is the same as \texttt{rk4(y,n,x1,h,derivs)} except that you must provide 4 scratch arrays t1-t4 of size n.
- \texttt{rk4f(y,n,x1,x2,ns,derivs)} uses ns fixed steps of a 4-th order Runge-Kutta integrator to advance the solution vector to x2 and return the values in y. Argument x2, is the final point, and \texttt{ns}, the number of steps to take.
- \texttt{rk4qc(y,n,x1,step,eps,yscal,derivs)} takes a 5-th order Runge-Kutta step with monitoring of local truncation to ensure accuracy and adjust stepsize.

The function takes two half steps and one full step and scales the difference in solutions at the final point. If the error is within \texttt{eps}, the step is taken and the result is returned. If the error is not within \texttt{eps}, the stepsize is decreased and the procedure is tried again until the desired accuracy is reached. Upon input, an trial step size must be given and upon return, an estimate of the next step size to use is returned as well as the step size which produced the desired accuracy.

The scaled error is computed as:
- \texttt{error = MAX(ABS((y2steps(i) - y1step(i))/yscal(i)))}
and this is compared against \texttt{eps}. If this is greater than \texttt{eps}, the step size is reduced accordingly to:
- \texttt{hnew = 0.9 * hdid * (error/eps)**(-1/4)}
If the error criterion is satisfied, then we check if the...
**CHAPTER 15. CHAPTER N**

++ step size was too fine and return a more efficient one. If
++ \spad{error > \spad{eps} * (6.0E-04)} then the next step size should be
++ Otherwise \spad{hnext = 4.0 * hdid} is returned.
++ A more detailed discussion of this and related topics can be
++ found in the book "Numerical Recipies" by W.Press, B.P. Flannery,
++
++ Argument \spad{step} is a record of 3 floating point
++ numbers \spad{(try , did , next)},
++ \spad{eps} is the required accuracy,
++ \spad{yscal} is the scaling vector for the difference in solutions.
++ On input, \spad{step.try} should be the guess at a step
++ size to achieve the accuracy.
++ On output, \spad{step.did} contains the step size which achieved the
++ accuracy and \spad{step.next} is the next step size to use.
++
++ \spad{rk4qc(y,n,x1,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,t7)} is the
++ same as \spad{rk4qc(y,n,x1,step,eps,yscal,derivs)} except that the user
++ must provide the 7 scratch arrays \spad{t1-t7} of size n.
++
++ \spad{rk4a(y,n,x1,x2,eps,h,ns,derivs)}
++ is a driver program which uses \spad{rk4qc} to integrate n ordinary
++ differential equations starting at x1 to x2, keeping the local
++ truncation error to within \spad{eps} by changing the local step size.
++ The scaling vector is defined as
++ \spad{yscal(i) = abs(y(i)) + abs(h*dydx(i)) + tiny}
++ where \spad{y(i)} is the solution at location x, \spad{dydx} is the
++ ordinary differential equation’s right hand side, h is the current
++ step size and \spad{tiny} is 10 times the
++ smallest positive number representable.
++
++ The user must supply an estimate for a trial step size and
++ the maximum number of calls to \spad{rk4qc} to use.
++ Argument x2 is the final point,
++ \spad{eps} is local truncation,
++ \spad{ns} is the maximum number of call to \spad{rk4qc} to use.

NumericalOrdinaryDifferentialEquations(): Exports == Implementation where
L ==> List
V ==> Vector
B ==> Boolean
I ==> Integer
E ==> OutputForm
NF ==> Float
NNI ==> NonNegativeInteger
VOID ==> Void
OFORM ==> OutputForm
RK4STEP ==> Record(try:NF, did:NF, next:NF)
Exports ==> with
--header definitions here

rk4 : (V NF,I,NF,NF, (V NF,V NF,NF) -> VOID) -> VOID
++ rk4(y,n,x1,h,derivs) uses a 4-th order Runge-Kutta method
++ to numerically integrate the ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector.
++ Argument y is a vector of initial conditions of length n which upon exit
++ contains the solution at \spad{x1 + h}, n is the number of dependent
++ variables, x1 is the initial point, h is the step size, and
++ \spad{derivs} is a function which computes the right hand side of the
++ ordinary differential equation.
++ For details, see \spadtype{NumericalOrdinaryDifferentialEquations}.

rk4 : (V NF,I,NF,NF, (V NF,V NF,NF) -> VOID
 ,V NF,V NF,V NF,V NF) -> VOID
++ \spad{rk4(y,n,x1,h,derivs,t1,t2,t3,t4)} is the same as
++ \spad{rk4a(y,n,x1,x2,eps,h,ns,derivs)} except that you must provide 4 scratch
++ arrays t1-t4 of size n.
++ For details, see \spad{NumericalOrdinaryDifferentialEquations}.

rk4a : (V NF,I,NF,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
++ rk4a(y,n,x1,x2,eps,h,ns,derivs) is a driver function for the
++ numerical integration of an ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector
++ using a 4-th order Runge-Kutta method.
++ For details, see \spad{NumericalOrdinaryDifferentialEquations}.

rk4qc : (V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID) -> VOID
++ rk4qc(y,n,x1,step,eps,yscal,derivs) is a subfunction for the
++ numerical integration of an ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector
++ using a 4-th order Runge-Kutta method.
++ This function takes a 5-th order Runge-Kutta step with monitoring
++ of local truncation to ensure accuracy and adjust stepsize.
++ For details, see \spad{NumericalOrdinaryDifferentialEquations}.

rk4f : (V NF,I,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
++ rk4f(y,n,x1,x2,ns,derivs) uses a 4-th order Runge-Kutta method
++ to numerically integrate the ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector.
++ Starting with y at x1, this function uses \spad{ns} fixed
++ steps of a 4-th order Runge-Kutta integrator to advance the
++ solution vector to x2 and return the values in y.
++ For details, see \spad{NumericalOrdinaryDifferentialEquations}. 2755
Implementation => add
--some local function definitions here
rk4qclocal : (V NF,V NF,I,NF,RK4STEP,NF,V NF,V NF,NF,V NF,NF) -> VOID
rk4local : (V NF,V NF,I,NF,NF,V NF,V NF,NF,V NF,NF) -> VOID
import OutputPackage
------------------------------------------------------------

rk4a(ystart,nvar,x1,x2,eps,htry,nstep,derivs) ==
y : V NF := new(nvar::NNI,0.0)
yscal : V NF := new(nvar::NNI,1.0)
dydx : V NF := new(nvar::NNI,0.0)
t1 : V NF := new(nvar::NNI,0.0)
t2 : V NF := new(nvar::NNI,0.0)
t3 : V NF := new(nvar::NNI,0.0)
t4 : V NF := new(nvar::NNI,0.0)
t5 : V NF := new(nvar::NNI,0.0)
t6 : V NF := new(nvar::NNI,0.0)
step : RK4STEP := [htry,0.0,0.0]
x : NF := x1
tiny : NF := 10.0**(-digits()+1)::I
m : I := nvar
outlist : L OFORM := [x::E,x::E,x::E]
i : I
iter : I
eps := 1.0/eps
for i in 1..m repeat
  y(i) := ystart(i)
for iter in 1..nstep repeat
  --compute the derivative
  derivs(dydx,y,x)
  --if overshoot, the set h accordingly
  if (x + step.try - x2) > 0.0 then
    step.try := x2 - x
  --find the correct scaling
  for i in 1..m repeat
    yscal(i) := abs(y(i)) + abs(step.try * dydx(i)) + tiny
  --take a quality controlled runge-kutta step
  rk4qclocal(y,dydx,nvar,x,step,eps,yscal,derivs
    ,t1,t2,t3,t4,t5,t6)
x := x + step.did
-- outlist.0 := x::E
-- outlist.1 := y(0)::E
-- outlist.2 := y(1)::E
-- output(blankSeparate(outlist)::E)
--check to see if done
if (x-x2) >= 0.0 then
leave
--next stepsize to use
    step.try := step.next
--end nstep repeat
if iter = (nstep+1) then
    output("ode: ERROR ")
    outlist.1 := nstep::E
    outlist.2 := " steps to small, last h = ":E
    outlist.3 := step.did::E
    output(blankSeparate(outlist))
    output(" y = ",y::E)
for i in 1..m repeat
    ystart(i) := y(i)

---------------------------------

rk4qc(y,n,x,step,eps,yscal,derivs) ==
t1 : V NF := new(n::NNI,0.0)
t2 : V NF := new(n::NNI,0.0)
t3 : V NF := new(n::NNI,0.0)
t4 : V NF := new(n::NNI,0.0)
t5 : V NF := new(n::NNI,0.0)
t6 : V NF := new(n::NNI,0.0)
t7 : V NF := new(n::NNI,0.0)
    derivs(t7,y,x)
    eps := 1.0/eps
    rk4qclocal(y,t7,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)
---------------------------------

rk4qc(y,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,dydx) ==
    derivs(dydx,y,x)
    eps := 1.0/eps
    rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)
---------------------------------

rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs
 ,t1,t2,t3,ysav,dysav,ytemp) ==
xsav : NF := x
h    : NF := step.try
fcor  : NF := 1.0/15.0
safety : NF := 0.9
grow  : NF := -0.20
shrink : NF := -0.25
errcon : NF := 0.6E-04 --(this is 4/safety)**(1/grow)
hh     : NF
errmax : NF
i      : I
m      : I := n
for i in 1..m repeat
dysav(i) := dydx(i)
ysav(i) := y(i)
--cut down step size till error criterion is met
repeat
--take two little steps to get to x + h
hh := 0.5 * h
rk4local(y sav, dysav, n, xsav, hh, ytemp, derivs, t1, t2, t3)
x := xsav + hh
derivs(dydx, ytemp, x)
rk4local(ytemp, dydx, n, x, hh, y, derivs, t1, t2, t3)
x := xsav + h
--take one big step get to x + h
rk4local(y sav, dysav, n, xsav, h, ytemp, derivs, t1, t2, t3)

--compute the maximum scaled difference
errmax := 0.0
for i in 1..m repeat
  ytemp(i) := y(i) - ytemp(i)
  errmax := max(errmax, abs(ytemp(i)/yscal(i)))
--scale relative to required accuracy
errmax := errmax * eps
--update integration steps size
if (errmax > 1.0) then
  h := safety * h * (errmax ** shrink)
else
  step.did := h
  if errmax > errcon then
    step.next := safety * h * (errmax ** grow)
  else
    step.next := 4 * h
  leave
--make fifth order with 4-th order error estimate
for i in 1..m repeat
  y(i) := y(i) + ytemp(i) * fcor

-------------------------------
rk4f(y, nvar, x1, x2, nstep, derivs) ==
yt : V NF := new(nvar::NNI, 0.0)
dyt : V NF := new(nvar::NNI, 0.0)
dym : V NF := new(nvar::NNI, 0.0)
dydx : V NF := new(nvar::NNI, 0.0)
ynew : V NF := new(nvar::NNI, 0.0)
h : NF := (x2-x1) / (nstep::NF)
x : NF := x1
i : I
j : I
-- start integrating
for i in 1..nstep repeat
    derivs(dydx,y,x)
    rk4local(y,dydx,nvar,x,h,y,derivs,yt,dyt,dym)
    x := x + h

--------------------------------------------------------

rk4(y,n,x,h,derivs) ==
    t1 : V NF := new(n::NNI,0.0)
    t2 : V NF := new(n::NNI,0.0)
    t3 : V NF := new(n::NNI,0.0)
    t4 : V NF := new(n::NNI,0.0)
    derivs(t1,y,x)
    rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)

------------------------------------------------------------

rk4(y,n,x,h,derivs,t1,t2,t3,t4) ==
    derivs(t1,y,x)
    rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)

------------------------------------------------------------

rk4local(y,dydx,n,x,h,yout,derivs,yt,dyt,dym) ==
    hh : NF := h*0.5
    h6 : NF := h/6.0
    xh : NF := x+hh
    m : I := n
    i : I
    -- first step
    for i in 1..m repeat
        yt(i) := y(i) + hh*dydx(i)
    -- second step
    derivs(dyt,yt,xh)
    for i in 1..m repeat
        yt(i) := y(i) + hh*dyt(i)
    -- third step
    derivs(dym,yt,xh)
    for i in 1..m repeat
        yt(i) := y(i) + h*dym(i)
        dym(i) := dyt(i) + dym(i)
    -- fourth step
    derivs(dyt,yt,x+h)
    for i in 1..m repeat
        yout(i) := y(i) + h6*( dydx(i) + 2.0*dym(i) + dyt(i) )


package NUMQUAD NumericalQuadrature

--- NumericalQuadrature.input ---

)set break resume
)sys rm -f NumericalQuadrature.output
)spool NumericalQuadrature.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalQuadrature
--E 1

)spool
)lisp (bye)

--- NumericalQuadrature.help ---

====================================================================
NumericalQuadrature examples
====================================================================

This suite of routines performs numerical quadrature using algorithms derived from the basic trapezoidal rule. Because the error term of this rule contains only even powers of the step size (for open and closed versions), fast convergence can be obtained if the integrand is sufficiently smooth.

Each routine returns a Record of type TrapAns, which contains
value Float: estimate of the integral
error Float: estimate of the error in the computation
totalpts Integer: total number of function evaluations
success Boolean: if the integral was computed within the user specified error criterion

To produce this estimate, each routine generates an internal sequence of sub-estimates, denoted by S(i), depending on the routine, to which the various convergence criteria are applied. The user must supply a relative accuracy, eps_r, and an absolute accuracy, eps_a. Convergence is obtained when either

\[ \text{ABS}(S(i) - S(i-1)) < \text{eps}_r \times \text{ABS}(S(i-1)) \]
\[ \text{or} \]
\[ \text{ABS}(S(i) - S(i-1)) < \text{eps}_a \]

are true statements.

The routines come in three families and three flavors:
- closed: romberg, simpson, trapezoidal
- open: rombergo, simpsono, trapezoidalo
- adaptive closed: aромберг, аsimpson, aтрапеzoidal

The S(i) for the trapezoidal family is the value of the integral using an equally spaced абсиска trapezoidal rule for that level of refinement.

The S(i) for the simpson family is the value of the integral using an equally spaced абсиска simpson rule for that level of refinement.

The S(i) for the romberg family is the estimate of the integral using an equally spaced абсиска romberg method. For the i-th level, this is an appropriate combination of all the previous trapezoidal estimates so that the error term starts with the 2*(i+1) power only.

The three families come in a closed version, where the formulas include the endpoints, an open version where the formulas do not include the endpoints and an adaptive version, where the user is required to input the number of subintervals over which the appropriate closed family integrator will apply with the usual convergence parameters for each subinterval. This is useful where a large number of points are needed only in a small fraction of the entire domain.

Each routine takes as arguments:
- f integrand
- a starting point
- b ending point
- eps_r relative error
- eps_a absolute error
- nmin refinement level when to start checking for convergence (> 1)
- nmax maximum level of refinement

The adaptive routines take as an additional parameter,
- nint, the number of independent intervals to apply a closed family integrator of the same name.
Note that closed family level $i$ uses $1 + 2^i$ points. Open family level $i$ uses $1 + 3^i$ points.

See Also:
- \texttt{show NumericalQuadrature}

---

**NumericalQuadrature (NUMQUAD)**

Exports:
- \texttt{aromberg}
- \texttt{asimpson}
- \texttt{atrapezoidal}
- \texttt{romberg}
- \texttt{rombergo}
- \texttt{simpson}
- \texttt{simpsono}
- \texttt{trapezoidal}
- \texttt{trapezoidalo}

---

\texttt{)abbrev package NUMQUAD NumericalQuadrature}

++ Author: Yurij A. Baransky
++ Date Created: October 90
++ Date Last Updated: October 90
++ Description:
++ This suite of routines performs numerical quadrature using
++ algorithms derived from the basic trapezoidal rule. Because
++ the error term of this rule contains only even powers of the
++ step size (for open and closed versions), fast convergence
++ can be obtained if the integrand is sufficiently smooth.
++
++ Each routine returns a Record of type TrapAns, which contains
++ value \texttt{Float}: estimate of the integral
++ error \texttt{Float}: estimate of the error in the computation
++ totalpts \texttt{Integer}: total number of function evaluations
++ success \texttt{Boolean}: if the integral was computed within the user
++ specified error criterion
To produce this estimate, each routine generates an internal
sequence of sub-estimates, denoted by $S(i)$, depending on the
routine, to which the various convergence criteria are applied.
The user must supply a relative accuracy, $\texttt{eps_r}$, and an absolute
accuracy, $\texttt{eps_a}$. Convergence is obtained when either:
\begin{itemize}
  \item $\texttt{ABS}(S(i) - S(i-1)) < \texttt{eps_r} \times \texttt{ABS}(S(i-1))$
  \item $\texttt{ABS}(S(i) - S(i-1)) < \texttt{eps_a}$
\end{itemize}
are true statements.

The routines come in three families and three flavors:
closed: romberg, simpson, trapezoidal
open: rombergo, simpsono, trapezoidalo
adaptive closed: aromberg, asimpson, atrapezoidal

The $S(i)$ for the trapezoidal family is the value of the
integral using an equally spaced absicca trapezoidal rule for
that level of refinement.

The $S(i)$ for the simpson family is the value of the integral
using an equally spaced absicca simpson rule for that level of
refinement.

The $S(i)$ for the romberg family is the estimate of the integral
using an equally spaced absicca romberg method. For
the $i$-th level, this is an appropriate combination of all the
previous trapezoidal estimates so that the error term starts
with the $2*(i+1)$ power only.

The three families come in a closed version, where the formulas
include the endpoints, an open version where the formulas do not
include the endpoints and an adaptive version, where the user
is required to input the number of subintervals over which the
appropriate closed family integrator will apply with the usual
convergence parameters for each subinterval. This is useful
where a large number of points are needed only in a small fraction
of the entire domain.

Each routine takes as arguments:
\begin{itemize}
  \item $f$ integrand
  \item a starting point
  \item b ending point
  \item $\texttt{eps_r}$ relative error
  \item $\texttt{eps_a}$ absolute error
  \item $\texttt{nmin}$ refinement level when to start checking for convergence ($> 1$)
  \item $\texttt{nmax}$ maximum level of refinement
\end{itemize}

The adaptive routines take as an additional parameter,
$nint$, the number of independent intervals to apply a closed
family integrator of the same name.
Notes:

Closed family level \( i \) uses \( 1 + 2^*i \) points.

Open family level \( i \) uses \( 1 + 3^*i \) points.

NumericalQuadrature(): Exports == Implementation where

\[
\begin{align*}
L \quad & \Rightarrow \text{List} \\
V \quad & \Rightarrow \text{Vector} \\
I \quad & \Rightarrow \text{Integer} \\
B \quad & \Rightarrow \text{Boolean} \\
E \quad & \Rightarrow \text{OutputForm} \\
F \quad & \Rightarrow \text{Float} \\
PI \quad & \Rightarrow \text{PositiveInteger} \\
OFORM \quad & \Rightarrow \text{OutputForm} \\
\end{align*}
\]

TrapAns \( \Rightarrow \text{Record(value:F, error:F, totalpts:I, success:B)} \)

Exports == with

aromberg : (F -> F,F,F,F,I,I,I) \( \rightarrow \) TrapAns

++ uses the adaptive romberg method to numerically integrate function
++ \( \text{spad}(\text{fn}) \) over the closed interval from \( \text{spad}(a) \) to \( \text{spad}(b) \),
++ with relative accuracy \( \text{spad(epsrel)} \) and absolute accuracy
++ \( \text{spad(epsabs)} \), with the refinement levels for convergence checking
++ vary from \( \text{spad(nmin)} \) to \( \text{spad(nmax)} \), and where \( \text{spad(nint)} \)
++ is the number of independent intervals to apply the integrator.
++ The value returned is a record containing the value of the integral,
++ the estimate of the error in the computation, the total number of
++ function evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \( \text{spadtype(NumericalQuadrature)} \) for details.

asimpson : (F -> F,F,F,F,I,I,I) \( \rightarrow \) TrapAns

++ adaptive simpson method to numerically integrate function \( \text{spad}(\text{fn}) \)
++ over the closed interval from \( \text{spad}(a) \) to \( \text{spad}(b) \), with relative
++ accuracy \( \text{spad(epsrel)} \) and absolute accuracy \( \text{spad(epsabs)} \), with the
++ refinement levels for convergence checking vary from \( \text{spad(nmin)} \)
++ to \( \text{spad(nmax)} \), and where \( \text{spad(nint)} \) is the number of independent
++ intervals to apply the integrator. The value returned is a record
++ containing the value of the integral, the estimate of the error in
++ the computation, the total number of function evaluations, and
++ either a boolean value which is true if the integral was computed
++ within the user specified error criterion.
++ See \( \text{spadtype(NumericalQuadrature)} \) for details.

atrapezoidal : (F -> F,F,F,F,I,I,I) \( \rightarrow \) TrapAns

++ adaptive trapezoidal method to numerically integrate function \( \text{spad}(\text{fn}) \)
++ over the closed interval from \( \text{spad}(a) \) to \( \text{spad}(b) \), with relative
++ accuracy \( \text{spad(epsrel)} \) and absolute accuracy \( \text{spad(epsabs)} \),
++ with the refinement levels for convergence checking vary from
++ \( \text{spad(nmin)} \) to \( \text{spad(nmax)} \), and where \( \text{spad(nint)} \) is the number
++ of independent intervals to apply the integrator. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

romberg  : (F -> F,F,F,F,F,I,I) -> TrapAns
++ romberg(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
++ method to numerically integrate function \spad{fn} over the closed
++ interval \spad{a} to \spad{b}, with relative accuracy \spad{epsrel}
++ and absolute accuracy \spad{epsabs}, with the refinement levels
++ for convergence checking vary from \spad{nmin} to \spad{nmax}.
++ The value returned is a record containing the value
++ of the integral, the estimate of the error in the computation, the
++ total number of function evaluations, and either a boolean value
++ which is true if the integral was computed within the user specified
++ error criterion. See \spadtype{NumericalQuadrature} for details.

simpson  : (F -> F,F,F,F,F,I,I) -> TrapAns
++ simpson(fn,a,b,epsrel,epsabs,nmin,nmax) uses the simpson
++ method to numerically integrate function \spad{fn} over the closed
++ interval \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

trapezoidal : (F -> F,F,F,F,F,I,I) -> TrapAns
++ trapezoidal(fn,a,b,epsrel,epsabs,nmin,nmax) uses the trapezoidal
++ method to numerically integrate function \spad{fn} over
++ the closed interval \spad{a} to \spad{b}, with relative accuracy
++ \spad{epsrel} and absolute accuracy \spad{epsabs}, with the
++ refinement levels for convergence checking vary
++ from \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the
++ estimate of the error in the computation, the total number of
++ function evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

rombergo : (F -> F,F,F,F,F,I,I) -> TrapAns
++ rombergo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
++ method to numerically integrate function \spad{fn} over
++ the open interval from \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

simpsono : (F -> F,F,F,F,F,I,I) -> TrapAns
++ simpsono(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
++ simpson method to numerically integrate function \spad{fn} over
++ the open interval from \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

trapezoidalo : (F -> F,F,F,F,F,I,I) -> TrapAns
++ trapezoidalo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
++ trapezoidal method to numerically integrate function \spad{fn}
++ over the open interval from \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

Implementation ==> add
trapclosed : (F -> F,F,F,F,I) -> F
trapopen : (F -> F,F,F,F,I) -> F
import OutputPackage

---------------------------------------------------
aromberg(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
ans : TrapAns
sum : F := 0.0
err : F := 0.0
pts : I := 1
done : B := true
hh : F := (b-a) / nint
x1 : F := a
x2 : F := a + hh
io : L OFORM := [x1::E,x2::E]
i : I
for i in 1..nint repeat
ans := romberg(func,x1,x2,epsrel,epsabs,nmin,nmax)
if (not ans.success) then
  io.1 := x1::E
  io.2 := x2::E
print blankSeparate cons("accuracy not reached in interval":E,io)
sum := sum + ans.value
err := err + abs(ans.error)
pts := pts + ans.totalpts-1
done := (done and ans.success)
x1 := x2
x2 := x2 + hh
return( [sum , err , pts , done] )

---------------------------------------------------
asimpson(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
ans : TrapAns
sum : F := 0.0
err : F := 0.0
pts : I := 1
done : B := true
hh : F := (b-a) / nint
x1 : F := a
x2 : F := a + hh
io : L OFORM := [x1::E,x2::E]
i : I
for i in 1..nint repeat
  ans := simpson(func,x1,x2,epsrel,epsabs,nmin,nmax)
  if (not ans.success) then
    io.1 := x1::E
    io.2 := x2::E
    print blankSeparate cons("accuracy not reached in interval":E,io)
  sum := sum + ans.value
  err := err + abs(ans.error)
  pts := pts + ans.totalpts-1
  done := (done and ans.success)
  x1 := x2
  x2 := x2 + hh
return( [sum , err , pts , done] )

---------------------------------------------------
atrapezoidal(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
ans : TrapAns
sum : F := 0.0
err : F := 0.0
pts : I := 1
i : I
done : B := true
hh : F := (b-a) / nint
x1 : F := a
x2 : F := a + hh
io : L OFORM := [x1::E,x2::E]
for i in 1..nint repeat
CHAPTER 15. CHAPTER N

ans := trapezoidal(func,x1,x2,epsrel,epsabs,nmin,nmax)
if (not ans.success) then
  io.1 := x1::E
  io.2 := x2::E
  print blankSeparate cons("accuracy not reached in interval":E,io)
  sum := sum + ans.value
  err := err + abs(ans.error)
  pts := pts + ans.totalpts-1
  done := (done and ans.success)
  x1 := x2
  x2 := x2 + hh
return( [sum , err , pts , done] )

---------------------------------------------------

romberg(func,a,b,epsrel,epsabs,nmin,nmax) ==
length : F := (b-a)
delta : F := length
newsum : F := 0.5 * length * (func(a)+func(b))
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
qx1 : F := newsum
table : V F := new((nmax+1)::PI,0.0)
n : I := 1
pts : I := 1
four : I
j : I
i : I
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapclosed(func,a,delta,oldsum,pts)
  newest := (4.0 * newsum - oldsum) / 3.0
\begin{verbatim}
four := 4
table(n) := newest
for j in 2..n repeat
  i := n+1-j
  four := four * 4
  table(i) := table(i+1) + (table(i+1)-table(i)) / (four-1)
if n > nmin then
  change := abs(table(1) - qx1)
  if change < abs(epsrel*qx1) then
    return( [table(1) , change , 2*pts+1 , true] )
  if change < epsabs then
    return( [table(1) , change , 2*pts+1 , true] )
oldsum := newsum
oldest := newest
delta := 0.5*delta
pts := 2*pts
qx1 := table(1)
return( [table(1) , 1.25*change , pts+1 ,false] )

simpson(func,a,b,epsrel,epsabs,nmin,nmax) ==
length : F := (b-a)
delta : F := length
newsum : F := 0.5*(b-a)*(func(a)+func(b))
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
n : I := 1
pts : I := 1
if (nmin < 2) then
  output("simpson: nmin to small (nmin > 1) nmin = ",nmin::E)
  return([0.0,0.0,0,false])
if (nmax < nmin) then
  output("simpson: nmax < nmin : nmax = ",nmax::E)
  output(" nmin = ",nmin::E)
  return([0.0,0.0,0,false])
if (a = b) then
  output("simpson: integration limits are equal = ",a::E)
  return([0.0,0.0,1,true])
if (epsrel < 0.0) then
  output("simpson: eps_r < 0.0 : eps_r = ",epsrel::E)
  return([0.0,0.0,0,false])
if (epsabs < 0.0) then
  output("simpson: eps_a < 0.0 : eps_a = ",epsabs::E)
  return([0.0,0.0,0,false])
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapclosed(func,a,delta,oldsum,pts)
\end{verbatim}
newest := (4.0 * newsum - oldsum) / 3.0
if n > nmin then
    change := abs(newest-oldest)
    if change < abs(epsrel*oldest) then
        return( [newest , 1.25*change , 2*pts+1 , true] )
    if change < epsabs then
        return( [newest , 1.25*change , 2*pts+1 , true] )
oldsum := newsum
oldest := newest
delta := 0.5*delta
pts := 2*pts
return( [newest , 1.25*change , pts+1 ,false] )

------------------------

trapezoidal(func,a,b,epsrel,epsabs,nmin,nmax) ==
    length : F := (b-a)
    delta : F := length
    newsum : F := 0.5*(b-a)*(func(a)+func(b))
    change : F := 0.0
    oldsum : F
    n : I := 1
    pts : I := 1
    if (nmin < 2) then
        output("trapezoidal: nmin to small (nmin > 1) nmin = ",nmin::E)
        return([0.0,0.0,0,false])
    if (nmax < nmin) then
        output("trapezoidal: nmax < nmin : nmax = ",nmax::E)
        output(" nmin = ",nmin::E)
        return([0.0,0.0,0,false])
    if (a = b) then
        output("trapezoidal: integration limits are equal = ",a::E)
        return([0.0,0.0,1,true])
    if (epsrel < 0.0) then
        output("trapezoidal: eps_r < 0.0 : eps_r = ",epsrel::E)
        return([0.0,0.0,0,false])
    if (epsabs < 0.0) then
        output("trapezoidal: eps_a < 0.0 : eps_a = ",epsabs::E)
        return([0.0,0.0,0,false])
for n in 1..nmax repeat
    oldsum := newsum
    newsum := trapclosed(func,a,delta,oldsum,pts)
    if n > nmin then
        change := abs(newsum-oldsum)
        if change < abs(epsrel*oldsum) then
            return( [newsum , 1.25*change , 2*pts+1 , true] )
        if change < epsabs then
            return( [newsum , 1.25*change , 2*pts+1 , true] )
delta := 0.5*delta
pts := 2*pts
return( [newsum , 1.25*change , pts+1 ,false] )

-----------------------------------------------------------------------------------
rombergo(func,a,b,epsrel,epsabs,nmin,nmax) ==
  length : F := (b-a)
delta : F := length / 3.0
newsum : F := length * func( 0.5*(a+b) )
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
qx1 : F := newsum
table : V F := new((nmax+1)::PI,0.0)
four : I
j : I
i : I
n : I := 1
pts : I := 1
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapopen(func,a,delta,oldsum,pts)
  newest := (9.0 * newsum - oldsum) / 8.0
  table(n) := newest
  nine := 9
  output(newest::E)
  for j in 2..n repeat
    i := n+1-j
    nine := nine * 9
    table(i) := table(i+1) + (table(i+1)-table(i)) / (nine-1)
  if n > nmin then
    change := abs(table(1) - qx1)
    if change < abs(epsrel*qx1) then
      return( [table(1) , 1.5*change , 3*pts , true] )
    if change < epsabs then
      return( [table(1) , 1.5*change , 3*pts , true] )
  output(table::E)
  oldsum := newsum
  oldest := newest
delta := delta / 3.0
pts := 3*pts
qx1 := table(1)
return( [table(1) , 1.5*change , pts ,false] )

-----------------------------------------------------------------------------------
simpsono(func,a,b,epsrel,epsabs,nmin,nmax) ==
  length : F := (b-a)
delta : F := length / 3.0
newsum : F := length * func( 0.5*(a+b) )
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
n : I := 1
pts : I := 1
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapopen(func,a,delta,oldsum,pts)
  newest := (9.0 * newsum - oldsum) / 8.0
  output(newest::E)
  if n > nmin then
    change := abs(newest - oldest)
    if change < abs(epsrel*oldest) then
      return( [newest , 1.5*change , 3*pts , true] )
    if change < epsabs then
      return( [newest , 1.5*change , 3*pts , true] )
  oldsum := newsum
  oldest := newest
  delta := delta / 3.0
  pts := 3*pts
return( [newest , 1.5*change , pts ,false] )

-------------------------------------------------------------------

trapezoidalo(func,a,b,epsrel,epsabs,nmin,nmax) ==
  length : F := (b-a)
  delta : F := length/3.0
  newsum : F := length*func( 0.5*(a+b) )
  change : F := 0.0
  pts : I := 1
  oldsum : F
  n : I
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapopen(func,a,delta,oldsum,pts)
  output(newsum::E)
  if n > nmin then
    change := abs(newsum-oldsum)
    if change < abs(epsrel*oldsum) then
      return([newsum , 1.5*change , 3*pts , true] )
    if change < epsabs then
      return([newsum , 1.5*change , 3*pts , true] )
  delta := delta / 3.0
  pts := 3*pts
return([newsum , 1.5*change , pts ,false] )

---------------------------------------------------------------

trapclosed(func,start,h,oldsum,numpoints) ==

package NCEP NumericComplexEigenPackage

--- NumericComplexEigenPackage.input ---

)set break resume
)sys rm -f NumericComplexEigenPackage.output
)spool NumericComplexEigenPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
CHAPTER 15. CHAPTER N

 NumericComplexEigenPackage

This package computes explicitly eigenvalues and eigenvectors of matrices with entries over the complex rational numbers. The results are expressed either as complex floating numbers or as complex rational numbers depending on the type of the precision parameter.

See Also:
  o )show NumericComplexEigenPackage

Exports:
  characteristicPolynomial  complexEigenvalues  complexEigenvectors

---

\textbf{NumericComplexEigenPackage (NCEP)}

Exports:  
characteristicPolynomial  complexEigenvalues  complexEigenvectors

---

\textbf{package NCEP NumericComplexEigenPackage ---

)abbrev package NCEP NumericComplexEigenPackage}
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: Spring 1991
++ Description:
++ This package computes explicitly eigenvalues and eigenvectors of
++ matrices with entries over the complex rational numbers.
++ The results are expressed either as complex floating numbers or as
++ complex rational numbers depending on the type of the precision parameter.

NumericComplexEigenPackage(Par) : C == T
where
Par : Join(Field,OrderedRing) -- Float or RationalNumber

SE ==> Symbol()
RN ==> Fraction Integer
I ==> Integer
NF ==> Float
CF ==> Complex Float
GRN ==> Complex RN
GI ==> Complex Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
MRN ==> Matrix RN

MCF ==> Matrix CF
MGRN ==> Matrix GRN
MCPar ==> Matrix Complex Par
SUPGRN ==> SparseUnivariatePolynomial GRN
outForm ==> Record(outval:Complex Par,outmult:Integer,outvect:List MCPar)

C == with
  characteristicPolynomial : MGRN -> Polynomial GRN
  ++ characteristicPolynomial(m) returns the characteristic polynomial
  ++ of the matrix m expressed as polynomial
  ++ over complex rationals with a new symbol as variable.
  -- while the function in EigenPackage returns Fraction P GRN.
  characteristicPolynomial : (MGRN,SE) -> Polynomial GRN
  ++ characteristicPolynomial(m,x) returns the characteristic polynomial
  ++ of the matrix m expressed as polynomial
  ++ over Complex Rationals with variable x.
  -- while the function in EigenPackage returns Fraction P GRN.
  complexEigenvalues : (MGRN,Par) -> List Complex Par
  ++ complexEigenvalues(m,eps) computes the eigenvalues of the matrix
  ++ m to precision eps. The eigenvalues are expressed as complex
  ++ floats or complex rational numbers depending on the type of
  ++ eps (float or rational).
  complexEigenvectors : (MGRN,Par) -> List(outForm)
  ++ complexEigenvectors(m,eps) returns a list of
  ++ records each one containing
  ++ a complex eigenvalue, its algebraic multiplicity, and a list of
++ associated eigenvectors. All these results
++ are computed to precision eps and are expressed as complex floats
++ or complex rational numbers depending on the type of
++ eps (float or rational).
T == add

import InnerNumericEigenPackage(GRN,Complex Par,Par)

characteristicPolynomial(m:MGRN) : Polynomial GRN ==
x:SE:=new()$SE
multivariate(charpol m, x)

characteristicPolynomial(A:MGRN,x:SE):Polynomial GRN ==
multivariate(charpol A, x)

complexEigenvalues(m:MGRN,eps:Par) : List Complex Par ==
solve1(charpol m, eps)

complexEigenvectors(m:MGRN,eps:Par) :List outForm ==
innerEigenvectors(m,eps,factor$ComplexFactorization(RN,SUPGRN))
NumericContinuedFraction helps convert floating point numbers to continued fractions.

See Also:
- )show NumericContinuedFraction

Exports:
- continuedFraction

---

package NCNTFRAC NumericContinuedFraction
++ Author: Clifton J. Williamson
++ Date Created: 12 April 1990
$$\textbf{Description:}$$

`\spadtype{NumericContinuedFraction}` provides functions for converting floating point numbers to continued fractions.

$$\text{NumericContinuedFraction}(F)$$: Exports == Implementation where

- `F : FloatingPointSystem`
- `CFC ==> ContinuedFraction Integer`
- `I ==> Integer`
- `ST ==> Stream I`

Exports ==> with

- `continuedFraction: F -> CFC`
  
  continuedFraction(f) converts the floating point number \spad{f} to a reduced continued fraction.

Implementation ==> add

```spad
  cfc: F -> ST
  cfc(a) == delay
    aa := wholePart a
    zero?(b := a - (aa :: F)) => concat(aa,empty()$ST)
    concat(aa,cfc inv b)

  continuedFraction a ==
    aa := wholePart a
    zero?(b := a - (aa :: F)) =>
      reducedContinuedFraction(aa,empty()$ST)
    if negative? b then (aa := aa - 1; b := b + 1)
      reducedContinuedFraction(aa,cfc inv b)
```

— NCNTFRAC.dotabb —

"NCNTFRAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCNTFRAC"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"NCNTFRAC" -> "FIELD"
"NCNTFRAC" -> "RADCAT"

— NumericRealEigenPackage.input —
NumericRealEigenPackage (NREP)

---

— NumericRealEigenPackage.help —

====================================================================
NumericRealEigenPackage examples
====================================================================

This package computes explicitly eigenvalues and eigenvectors of matrices with entries over the Rational Numbers. The results are expressed as floating numbers or as rational numbers depending on the type of the parameter Par.

See Also:
- )show NumericRealEigenPackage

---

NREP

ALIST
Exports:
characteristicPolynomial realEigenvalues realEigenvectors

— package NREP NumericRealEigenPackage —

)abbrev package NREP NumericRealEigenPackage
++ Author:P. Gianni
++ Date Created:Summer 1990
++ Date Last Updated:Spring 1991
++ Description:
++ This package computes explicitly eigenvalues and eigenvectors of
++ matrices with entries over the Rational Numbers.
++ The results are expressed as floating numbers or as rational numbers
++ depending on the type of the parameter Par.

NumericRealEigenPackage(Par) : C == T
where
  Par : Join(Field,OrderedRing) -- Float or RationalNumber

  SE ==> Symbol()
  RN ==> Fraction Integer
  I ==> Integer
  NF ==> Float
  CF ==> Complex Float
  GRN ==> Complex RN
  GI ==> Complex Integer
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  MRN ==> Matrix RN

  MPar ==> Matrix Par
  outForm ==> Record(outval:Par,outmult:Integer,outvect:List MPar)

C == with
  characteristicPolynomial : MRN -> Polynomial RN
    ++ characteristicPolynomial(m) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over RN with a new symbol as variable.
    ++ while the function in EigenPackage returns Fraction P RN.
  characteristicPolynomial : (MRN,SE) -> Polynomial RN
    ++ characteristicPolynomial(m,x) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over RN with variable x.
    ++ while the function in EigenPackage returns
    ++ Fraction P RN.
  realEigenvalues : (MRN,Par) -> List Par
    ++ realEigenvalues(m,eps) computes the eigenvalues of the matrix
    ++ m to precision eps. The eigenvalues are expressed as floats or
    ++ rational numbers depending on the type of eps (float or rational).
realEigenvectors : (MRN,Par) -> List(outForm)
++ realEigenvectors(m,eps) returns a list of
++ records each one containing
++ a real eigenvalue, its algebraic multiplicity, and a list of
++ associated eigenvectors. All these results
++ are computed to precision eps as floats or rational
++ numbers depending on the type of eps.

T == add

import InnerNumericEigenPackage(RN, Par, Par)

characteristicPolynomial(m:MRN) : Polynomial RN ==
  x:SE:=new()$SE
  multivariate(charpol(m),x)

---- characteristic polynomial of a matrix A ----
characteristicPolynomial(A:MRN,x:SE):Polynomial RN ==
  multivariate(charpol(A),x)

realEigenvalues(m:MRN,eps:Par) : List Par ==
  solve1(charpol m, eps)

realEigenvectors(m:MRN,eps:Par) :List outForm ==
  innerEigenvectors(m,eps,factor$GenUFactorize(RN))
---

__NumericTubePlot.help__

====================================================================

**NumericTubePlot examples**

====================================================================

Package for constructing tubes around 3-dimensional parametric curves.

See Also:
- `)`show NumericTubePlot

---

**NumericTubePlot (NUMTUBE)**

Exports:
- `tube`

---

package NUMTUBE NumericTubePlot ---
)abbrev package NUMTUBE NumericTubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.

NumericTubePlot(Curve): Exports == Implementation where
    Curve : PlottableSpaceCurveCategory
    B ==> Boolean
    I ==> Integer
    SF ==> DoubleFloat
    L ==> List
    S ==> String
    SEG ==> Segment
    Pt ==> Point SF
    TUBE ==> TubePlot Curve
    Triad ==> Record(tang:Pt,norm:Pt,bin:Pt)

Exports ==> with
    tube: (Curve,SF,I) -> TUBE
    ++ tube(c,r,n) creates a tube of radius r around the curve c.

Implementation ==> add
    import TubePlotTools

    LINMAX := convert(0.995)@SF
    XHAT := point(1,0,0,0)
    YHAT := point(0,1,0,0)
    PREV0 := point(1,1,0,0)
    PREV := PREV0

    colinearity: (Pt,Pt) -> SF
    colinearity(x,y) == dot(x,y)**2/(dot(x,x) * dot(y,y))

    orthog: (Pt,Pt) -> Pt
    orthog(x,y) ==
        if colinearity(x,y) > LINMAX then y := PREV
        if colinearity(x,y) > LINMAX then
            y := (colinearity(x,XHAT) < LINMAX => XHAT; YHAT)
        a := -dot(x,y)/dot(x,x)
        PREV := a*x + y

    poTriad:(Pt,Pt,Pt) -> Triad
    poTriad(pl,po,pr) ==
        -- use divided difference for t.
        t := unitVector(pr - pl)
        -- compute n as orthogonal to t in plane containing po.
        pol := pl - po
        n := unitVector orthog(t,pol)
[t,n,cross(t,n)]

curveTriads: L Pt -> L Triad
curveTriads 1 ==
  (k := #l) < 2 => error "Need at least 2 points to specify a curve"
  PREV := PREV0
  k = 2 =>
    t := unitVector(second l - first l)
    n := unitVector(t - XHAT)
    b := cross(t,n)
    triad : Triad := [t,n,b]
    [triad,triad]
    -- compute interior triads using divided differences
    midtriads : L Triad :=
      [poTriad(pl,po,pr) for pl in l for po in rest l _
       for pr in rest rest l]
    -- compute first triad using a forward difference
    x := first midtriads
    t := unitVector(second l - first l)
    n := unitVector orthog(t,x.norm)
    begtriad : Triad := [t,n,cross(t,n)]
    -- compute last triad using a backward difference
    x := last midtriads
    -- efficiency!!
    t := unitVector(l.k - l.(k-1))
    n := unitVector orthog(t,x.norm)
    endtriad : Triad := [t,n,cross(t,n)]
    concat(begtriad,concat(midtriads,endtriad))

curveLoops: (L Pt,SF,I) -> L L Pt
curveLoops(pts,r,nn) ==
  triads := curveTriads pts
  cosSin := cosSinInfo nn
  loops : L L Pt := nil()
  for pt in pts for triad in triads repeat
    n := triad.norm; b := triad.bin
    loops := concat(loopPoints(pt,n,b,r,cosSin),loops)
  reverse_! loops

tube(curve,r,n) ==
  n < 3 => error "tube: n should be at least 3"
  brans := listBranches curve
  loops : L L Pt := nil()
  for bran in brans repeat
    loops := concat(loops,curveLoops(bran,r,n))
tube(curve,loops,false)
— NUMTUBE.dotab —

"NUMTUBE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMTUBE"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"NUMTUBE" -> "FIELD"
"NUMTUBE" -> "RADCAT"
"NUMTUBE" -> "FLAGG"
"NUMTUBE" -> "FLAGG-"
Chapter 16

Chapter 0

package OCTCT2 OctonionCategoryFunctions2

— OctonionCategoryFunctions2.input —

)set break resume
)sys rm -f OctonionCategoryFunctions2.output
)spool OctonionCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OctonionCategoryFunctions2
--E 1

)spool
)lisp (bye)

— OctonionCategoryFunctions2.help —

====================================================================
OctonionCategoryFunctions2 examples
====================================================================

OctonionCategoryFunctions2 implements functions between two octonion domains defined over different rings. The function map is used to coerce between octonion types.

2787
See Also:
- )show OctonionCategoryFunctions2

OctonionCategoryFunctions2 (OCTCT2)

Exports:
map

— package OCTCT2 OctonionCategoryFunctions2 —

)abbrev package OCTCT2 OctonionCategoryFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 10 September 1990
++ Date Last Updated: 10 September 1990
++ Description:
++ OctonionCategoryFunctions2 implements functions between
two octonion domains defined over different rings.
++ The function map is used to coerce between octonion types.

OctonionCategoryFunctions2(OR,R,OS,S) : Exports ==
Implementation where
R : CommutativeRing
S : CommutativeRing
OR : OctonionCategory R
OS : OctonionCategory S
Exports == with
map: (R -> S, OR) -> OS
++ map(f,u) maps f onto the component parts of the octonion
++ u.
Implementation == add
map(fn : R -> S, u : OR): OS ==
octon(fn real u, fn imagi u, fn imagj u, fn imagk u, _
fn imagE u, fn imagI u, fn imagJ u, fn imagK u)

package ODEINT ODEIntegration

ODEIntegration provides an interface to the integrator.
This package is intended for use by the differential equations
solver but not at top-level.
ODEIntegration (ODEINT)

Exports:
  diff  expint  int

— package ODEINT ODEIntegration —

)abbrev package ODEINT ODEIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 November 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spadtype{ODEIntegration} provides an interface to the integrator.
++ This package is intended for use
++ by the differential equations solver but not at top-level.

ODEIntegration(R, F): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
        LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
          PrimitiveFunctionCategory)

  Q ==> Fraction Integer
  UQ ==> Union(Q, "failed")
  SY ==> Symbol
  K ==> Kernel F
  P ==> SparseMultivariatePolynomial(R, K)
  REC ==> Record(coef:Q, logand:F)
Exports ==> with
int : (F, SY) -> F
  ++ int(f, x) returns the integral of f with respect to x.
expint : (F, SY) -> F
  ++ expint(f, x) returns e^{\text{the integral of f with respect to x}}.
diff : SY -> (F -> F)
  ++ diff(x) returns the derivation with respect to x.

Implementation ==> add
import FunctionSpaceIntegration(R, F)
import ElementaryFunctionStructurePackage(R, F)
isQ : List F -> UQ
isQlog: F -> Union(REC, "failed")
mkprod: List REC -> F
diff x == (f1:F):F +-> differentiate(f1, x)

-- This is the integration function to be used for quadratures
int(f, x) ==
  (u := integrate(f, x)) case F => u::F
  first(u::List(F))

-- mkprod([q1, f1],...,[qn,fn]) returns */(fi^qi) but groups the
-- qi having the same denominator together
mkprod l ==
  empty? l => 1
  rec := first l
  d := denom(rec.coef)
  ll := select((z1:REC):Boolean +-> denom(z1.coef) = d, l)
  nthRoot(*/[r.logand ** numer(r.coef) for r in ll], d) *
  mkprod setDifference(l, ll)

-- computes exp(int(f,x)) in a non-naive way
expint(f, x) ==
  a := int(f, x)
  (u := validExponential(tower a, a, x)) case F => u::F
da := denom a
l :=
  (v := isPlus(na := numer a)) case List(P) => v::List(P)
  [na]
  exponent:P := 0
lrec:List(REC) := empty()
for term in l repeat
  if (w := isQlog(term / da)) case REC then
    lrec := concat(w::REC, lrec)
  else
    exponent := exponent + term
  mkprod(lrec) * exp(exponent / da)
-- checks if all the elements of l are rational numbers, returns their product
isQ l ==
  prod:Q := 1
  for x in l repeat
    (u := retractIfCan(x)@UQ) case "failed" => return "failed"
    prod := prod * u::Q
  prod

-- checks if a non-sum expr is of the form c * log(g) for a rational number c
isQlog f ==
  is?(f, "log":SY) => [1, first argument(retract(f)@K)]
  (v := isTimes f) case List(F) and (#(l := v::List(F)) <= 3) =>
    l := reverse_! sort_! l
    is?(first l, "log":SY) and ((u := isQ rest l) case Q) =>
      [u::Q, first argument(retract(first(l))@K)]
    "failed"
  "failed"

package ODETOOLS ODETools

— ODETools.input —

)set break resume
)sys rm -f ODETools.output
)spool ODETools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ODETools
--E 1


---

### ODETools (ODETOOLS)

Exports:
- particularSolution
- variationOfParameters
- wronskianMatrix

---

```plaintext
)abbrev package ODETOOLS ODETools
++ Author: Manuel Bronstein
++ Date Created: 20 March 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spad{ODETools} provides tools for the linear ODE solver.
```
ODETools(F, LODO): Exports == Implementation where
N ==> NonNegativeInteger
L ==> List F
V ==> Vector F
M ==> Matrix F
F: Field
LODO: LinearOrdinaryDifferentialOperatorCategory F
Exports ==> with
wronskianMatrix: L -> M
++ wronskianMatrix([f1,...,fn]) returns the \(\text{spad{n x n}}\) matrix \(m\)
++ whose \(i\)'th row is \(\text{spad{[f1}^{(i-1)},...,fn^{(i-1)]}}\).
wronskianMatrix: (L, N) -> M
++ wronskianMatrix([f1,...,fn], q, D) returns the \(\text{spad{q x n}}\) matrix \(m\)
++ whose \(i\)'th row is \(\text{spad{[f1}^{(i-1)},...,fn^{(i-1)]}}\).
variationOfParameters: (LODO, F, L) -> Union(V, "failed")
++ variationOfParameters(op, g, [f1,...,fm])
++ returns \(\text{spad{[u1},...,um]}\) such that a particular solution of the
++ equation \(\text{spad{op y = g}}\) is \(\text{spad{f1 int(u1) + ... + fm int(um)}}\)
++ where \(\text{spad{[f1},...,fm]}\) are linearly independent and \(\text{spad{op(fi)=0}}\).
++ The value "failed" is returned if \(\text{spad{m < n}}\) and no particular
++ solution is found.
partialSolution: (LODO, F, L, F -> F) -> Union(F, "failed")
++ particularSolution(op, g, [f1,...,fm], I) returns a particular
++ solution \(h\) of the equation \(\text{spad{op y = g}}\) where \(\text{spad{[f1},...,fm]}\)
++ are linearly independent and \(\text{spad{op(fi)=0}}\).
++ The value "failed" is returned if no particular solution is found.
++ Note that the method of variations of parameters is used.
Implementation ==> add
import LinearSystemMatrixPackage(F, V, V, M)
diff := D()$LODO
wronskianMatrix l == wronskianMatrix(l, #l)
wronskianMatrix(l, q) ==
v:V := vector l
m:M := zero(q, #v)
for i in minRowIndex m .. maxRowIndex m repeat
  setRow_!(m, i, v)
  v := map_!((f1:F):F +-> diff f1, v)
m
variationOfParameters(op, g, b) ==
empty? b => "failed"
v:V := new(n := degree op, 0)
qsetelt_!(v, maxIndex v, g / leadingCoefficient op)
partialSolution(wronskianMatrix(b, n), v)
particularSolution(op, g, b, integration) ==
    zero? g => 0
    (sol := variationOfParameters(op, g, b)) case "failed" => "failed"
    ans:F := 0
    for f in b for i in minIndex(s := sol::V) .. repeat
        ans := ans + integration(qelt(s, i)) * f
    ans

package ARRAY12 OneDimensionalArrayFunctions2

--S 1 of 1
)show OneDimensionalArrayFunctions2
--E 1

)spool )lisp (bye)

=== OneDimensionalArrayFunctions2.examples ===
This package provides tools for operating on one-dimensional arrays with unary and binary functions involving different underlying types.

See Also:
- )show OneDimensionalArrayFunctions2

---

OneDimensionalArrayFunctions2 (ARRAY12)

Exports:
mmap reducescanscan

--- package ARRAY12 OneDimensionalArrayFunctions2 ---

)abbrev package ARRAY12 OneDimensionalArrayFunctions2
++ Description:
++ This package provides tools for operating on one-dimensional arrays
++ with unary and binary functions involving different underlying types

OneDimensionalArrayFunctions2(A, B): Exports == Implementation where
A, B: Type

VA => OneDimensionalArray A
VB => OneDimensionalArray B
O2 => FiniteLinearAggregateFunctions2(A, VA, B, VB)

Exports == with
scan : ((A, B) -> B, VA, B) -> VB
++ scan(f,a,r) successively applies
++ \spad{reduce(f,x,r)} to more and more leading sub-arrays
++ x of one-dimensional array \spad{a}.
++ More precisely, if \spad{a} is \spad{[[a1,a2,...]]}, then
++ \spad{scan(f,a,r)} returns
++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
++
++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
++X adder(a:Integer,b:Integer):Integer == a+b
++X scan(adder,[i for i in 1..10],0)$T1

reduce : ((A, B) -> B, VA, B) -> B
++ reduce(f,a,r) applies function f to each
++ successive element of the
++ one-dimensional array \spad{a} and an accumulant initialized to r.
++ For example, \spad{reduce(_+$Integer,[1,2,3],0)}
++ does \spad{3+(2+(1+0))}. Note that third argument r
++ may be regarded as the identity element for the function f.
++
++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
++X adder(a:Integer,b:Integer):Integer == a+b
++X reduce(adder,[i for i in 1..10],0)$T1

map : (A -> B, VA) -> VB
++ map(f,a) applies function f to each member of one-dimensional array
++ \spad{a} resulting in a new one-dimensional array over a
++ possibly different underlying domain.
++
++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
++X map(x+->x+2,[i for i in 1..10])$T1

Implementation ==> add
map(f, v)  == map(f, v)$O2
scan(f, v, b)  == scan(f, v, b)$O2
reduce(f, v, b)  == reduce(f, v, b)$O2

ARRAY12.dotabb

"ARRAY12" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ARRAY12"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARRAY12" -> "A1AGG"
package ONECOMP2 OnePointCompletionFunctions2

--- OnePointCompletionFunctions2.input ---

)set break resume
)sys rm -f OnePointCompletionFunctions2.output
)spool OnePointCompletionFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OnePointCompletionFunctions2
--E 1

)spool
)lisp (bye)

-----

--- OnePointCompletionFunctions2.help ---

====================================================================
OnePointCompletionFunctions2 examples
====================================================================

Lifting of maps to one-point completions.

See Also:
  o )show OnePointCompletionFunctions2

-----
OnePointCompletionFunctions2 (ONECOMP2)

Exports:
map

— package ONECOMP2 OnePointCompletionFunctions2 —

)abbrev package ONECOMP2 OnePointCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to one-point completions.

OnePointCompletionFunctions2(R, S): Exports == Implementation where
R, S: SetCategory

OPR ==> OnePointCompletion R
OPS ==> OnePointCompletion S

Exports ==> with
map: (R -> S, OPR) -> OPS
++ map(f, r) lifts f and applies it to r, assuming that
++ f(infinity) = infinity.
map: (R -> S, OPR, OPS) -> OPS
++ map(f, r, i) lifts f and applies it to r, assuming that
++ f(infinity) = i.

Implementation ==> add
map(f, r) == map(f, r, infinity())

map(f, r, i) ==
(u := retractIfCan r) case R => (f(u::R)::OPS
i

|---|
package OMPKG OpenMathPackage

OpenMathPackage examples
OpenMathPackage provides some simple utilities to make reading OpenMath objects easier.

See Also:
  o )show OpenMathPackage
OpenMathPackage (OMPKG)

Exports:
OMlistCDs  OMread  OMreadFile  OMreadStr  OMsupportsCD?
OMlistSymbols  OMsupportsSymbol?  OMunhandledSymbol

package OMPKG OpenMathPackage

)abbrev package OMPKG OpenMathPackage
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathPackage} provides some simple utilities
++ to make reading OpenMath objects easier.

OpenMathPackage(): with
  OMread : OpenMathDevice -> Any
    ++ OMread(dev) reads an OpenMath object from \axiom{dev} and passes it
    ++ to AXIOM.
  OMreadFile : String -> Any
    ++ OMreadFile(f) reads an OpenMath object from \axiom{f} and passes it
    ++ to AXIOM.
  OMreadStr : String -> Any
    ++ OMreadStr(f) reads an OpenMath object from \axiom{f} and passes it
    ++ to AXIOM.
  OMlistCDs : () -> List(String)
    ++ OMlistCDs() lists all the CDs supported by AXIOM.
  OMlistSymbols : String -> List(String)
    ++ OMlistSymbols(cd) lists all the symbols in \axiom{cd}.
  OMsupportsCD? : String -> Boolean
    ++ OMsupportsCD?(cd) returns true if AXIOM supports \axiom{cd}, false
    ++ otherwise.
  OMsupportsSymbol? : (String, String) -> Boolean
    ++ OMsupportsSymbol?(s,cd) returns true if AXIOM supports symbol \axiom{s}
    ++ from CD \axiom{cd}, false otherwise.
  OMunhandledSymbol : (String, String) -> Exit
    ++ OMunhandledSymbol(s,cd) raises an error if AXIOM reads a symbol which it
    ++ is unable to handle. Note that this is different from an unexpected
++ symbol.
== add
import OpenMathEncoding
import OpenMathDevice
import String

OMunhandledSymbol(u, v) ==
    error concat ["AXIOM is unable to process the symbol ", u, " from CD ", v, "."]

OMread(dev: OpenMathDevice): Any ==
    interpret(OM_-READ(dev)$Lisp :: InputForm)

OMreadFile(filename: String): Any ==
    dev := OMopenFile(filename, "r", OMencodingUnknown())
    res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
    OMclose(dev)
    res

OMreadStr(str: String): Any ==
    strp := OM_-STRINGTOSTRINGPTR(str)$Lisp
    dev := OMopenString(strp pretend String, OMencodingUnknown())
    res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
    OMclose(dev)
    res

OMlistCDs(): List(String) ==
    OM_-LISTCDS()$Lisp pretend List(String)

OMlistSymbols(cd: String): List(String) ==
    OM_-LISTSYMBOLS(cd)$Lisp pretend List(String)

import SExpression

OMsupportsCD?(cd: String): Boolean ==
    not null? OM_-SUPPORTSCD(cd)$Lisp

OMsupportsSymbol?(cd: String, name: String): Boolean ==
    not null? OM_-SUPPORTSSYMBOL(cd, name)$Lisp

OMPKG.dotabb

"OMPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMPKG"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMPKG" -> "STRING"
package OMSERVER OpenMathServerPackage

--- OpenMathServerPackage.input ---

)set break resume
)sys rm -f OpenMathServerPackage.output
)spool OpenMathServerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathServerPackage
--E 1

)spool
)lisp (bye)

---

--- OpenMathServerPackage.help ---

====================================================================
OpenMathServerPackage examples
====================================================================

OpenMathServerPackage provides the necessary operations to run Axiom as an OpenMath server, reading/writing objects to/from a port. Please note the facilities available here are very basic. The idea is that a user calls e.g. Omserve(4000,60) and then another process sends OpenMath objects to port 4000 and reads the result.

See Also:
  o )show OpenMathServerPackage
OpenMathServerPackage (OMSERVER)

Exports:
OMreceive  OMsend  OMserve

--- package OMSERVER OpenMathServerPackage ---

)abbrev package OMSERVER OpenMathServerPackage
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathServerPackage} provides the necessary
++ operations to run AXIOM as an OpenMath server, reading/writing objects
++ to/from a port. Please note the facilities available here are very basic.
++ The idea is that a user calls e.g. \axiom{Omserve(4000,60)} and then
++ another process sends OpenMath objects to port 4000 and reads the result.

OpenMathServerPackage(): with
  OMreceive : OpenMathConnection -> Any
  ++ OMreceive(c) reads an OpenMath object from connection \axiom{c} and
  ++ returns the appropriate AXIOM object.
  OMsend   : (OpenMathConnection, Any) -> Void
  ++ OMsend(c,u) attempts to output \axiom{u} on \axiom{c} in OpenMath.
  OMserve  : (SingleInteger, SingleInteger) -> Void
  ++ OMserve(portnum,timeout) puts AXIOM into server mode on port number
  ++ \axiom{portnum}. The parameter \axiom{timeout} specifies the timeout
  ++ period for the connection.
  == add
  import OpenMathDevice
  import OpenMathConnection
  import OpenMathPackage

OMreceive(conn: OpenMathConnection): Any ==
  dev: OpenMathDevice := OMconnInDevice(conn)
  OMsetEncoding(dev, OMencodingUnknown);
OMread(dev)

OMsend(conn: OpenMathConnection, value: Any): Void ==
  dev: OpenMathDevice := OMconnOutDevice(conn)
  OMsetEncoding(dev, OMencodingXML);
  --retractable?(value)$AnyFunctions1(Expression Integer) =>
  --  OMwrite(dev, retract(value)$AnyFunctions1(Expression Integer), true)
  retractable?(value)$AnyFunctions1(Integer) =>
    OMwrite(dev, retract(value)$AnyFunctions1(Integer), true)
  retractable?(value)$AnyFunctions1(Float) =>
    OMwrite(dev, retract(value)$AnyFunctions1(Float), true)
  retractable?(value)$AnyFunctions1(SingleInteger) =>
    OMwrite(dev, retract(value)$AnyFunctions1(SingleInteger), true)
  retractable?(value)$AnyFunctions1(DoubleFloat) =>
    OMwrite(dev, retract(value)$AnyFunctions1(DoubleFloat), true)
  retractable?(value)$AnyFunctions1(String) =>
    OMwrite(dev, retract(value)$AnyFunctions1(String), true)

OMserve(portNum: SingleInteger, timeout: SingleInteger): Void ==
  conn: OpenMathConnection := OMmakeConn(timeout)
  OMbindTCP(conn, portNum)
  val: Any
  while true repeat
    val := OMreceive(conn)
    OMsend(conn, val)

— OMSERVER.dotabb —

"OMSERVER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMSERVER"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMSERVER" -> "STRING"

package OPQUERY OperationsQuery

— OperationsQuery.input —

)set break resume
)sys rm -f OperationsQuery.output
)spool OperationsQuery.output
)set message test on
OperationsQuery (OPQUERY)

Exports:
getDatabase

— package OPQUERY OperationsQuery —

)abbrev package OPQUERY OperationsQuery
++ Description:
++ This package exports tools to create AXIOM Library information databases.

OperationsQuery(): Exports == Implementation where
Exports == with
  getDatabase: String -> Database(IndexCard)
  ++ getDatabase("char") returns a list of appropriate entries in the
  ++ browser database. The legal values for "char" are "o" (operations),
  ++ "k" (constructors), "d" (domains), "c" (categories) or "p" (packages).
Implementation == add
  getDatabase(s) == getBrowseDatabase(s)$Lisp

— OPQUERY.dotabb —

"OPQUERY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OPQUERY"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"OPQUERY" -> "ORDSET"

package ORDCOMP2 OrderedCompletionFunctions2

— OrderedCompletionFunctions2.input —

)set break resume
)sys rm -f OrderedCompletionFunctions2.output
)spool OrderedCompletionFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show OrderedCompletionFunctions2
--E 1

)spool
)lisp (bye)

— OrderedCompletionFunctions2.help —
OrderedCompletionFunctions2 examples

Lifting of maps to ordered completions.

See Also:
- )show OrderedCompletionFunctions2

---

OrderedCompletionFunctions2 (ORDCOMP2)

Exports:
map

--- package ORDCOMP2 OrderedCompletionFunctions2 ---

)abbrev package ORDCOMP2 OrderedCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to ordered completions.

OrderedCompletionFunctions2(R, S):Exports == Implementation where
  R, S: SetCategory
  ORR ==> OrderedCompletion R
  ORS ==> OrderedCompletion S

Exports == with
  map: (R -> S, ORR) -> ORS
++ map(f, r) lifts f and applies it to r, assuming that
++ f(plusInfinity) = plusInfinity and that
++ f(minusInfinity) = minusInfinity.
map: (R -> S, Orr, ORS) -> ORS
++ map(f, r, p, m) lifts f and applies it to r, assuming that
++ f(plusInfinity) = p and that f(minusInfinity) = m.

Implementation ==> add
map(f, r) == map(f, r, plusInfinity(), minusInfinity())

map(f, r, p, m) ==
zero?(n := whatInfinity r) => (f retract r)::ORS
-- one? n => p
(n = 1) => p
m

---

-- ORDCOMP2.dotabb --

"ORDCOMP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORDCOMP2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ORDCOMP2" -> "BASTYPE"
"ORDCOMP2" -> "KOERCE"

---

package ORDFUNS OrderingFunctions

--- OrderingFunctions.input ---

)set break resume
)sys rm -f OrderingFunctions.output
)spool OrderingFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderingFunctions
--E 1

)spool
)lisp (bye)
This package provides ordering functions on vectors which are suitable parameters for OrderedDirectProduct.

See Also:
o )show OrderingFunctions
T == with
  pureLex : (VS,VS) -> Boolean
  ++ pureLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the lexicographic ordering.
  totalLex : (VS,VS) -> Boolean
  ++ totalLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the ordering which is total degree refined by
  ++ lexicographic ordering.
  reverseLex : (VS,VS) -> Boolean
  ++ reverseLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the ordering which is total degree refined by
  ++ the reverse lexicographic ordering.

C == add
  n:NonNegativeInteger:=dim

  -- pure lexicographical ordering
  pureLex(v1:VS,v2:VS) : Boolean ==
    for i in 1..n repeat
      if qelt(v1,i) < qelt(v2,i) then return true
      if qelt(v2,i) < qelt(v1,i) then return false
    false

  -- total ordering refined with lex
  totalLex(v1:VS,v2:VS) :Boolean ==
    n1:S:=0
    n2:S:=0
    for i in 1..n repeat
      n1:= n1+qelt(v1,i)
      n2:=n2+qelt(v2,i)
    n1<n2 => true
    n2<n1 => false
    for i in 1..n repeat
      if qelt(v1,i) < qelt(v2,i) then return true
      if qelt(v2,i) < qelt(v1,i) then return false
    false

  -- reverse lexicographical ordering
  reverseLex(v1:VS,v2:VS) :Boolean ==
    n1:S:=0
    n2:S:=0
    for i in 1..n repeat
      n1:= n1+qelt(v1,i)
      n2:=n2+qelt(v2,i)
    n1<n2 => true
    n2<n1 => false
    for i in reverse(1..n) repeat
      if qelt(v2,i) < qelt(v1,i) then return true
      if qelt(v1,i) < qelt(v2,i) then return false
false

---

--- ORDFUNS.dotabb ---

"ORDFUNS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORDFUNS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ORDFUNS" -> "IVECTOR"

---

package ORTHPOL OrthogonalPolynomialFunctions

--- OrthogonalPolynomialFunctions.input ---

)set break resume
/sys rm -f OrthogonalPolynomialFunctions.output
/spool OrthogonalPolynomialFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrthogonalPolynomialFunctions
--E 1

)spool
)lisp (bye)

---

--- OrthogonalPolynomialFunctions.help ---

====================================================================
OrthogonalPolynomialFunctions examples
====================================================================

This package provides orthogonal polynomials as functions on a ring.

See Also:
  o )show OrthogonalPolynomialFunctions
OrthogonalPolynomialFunctions (ORTHPOL)

Exports:
chebyshevT chebyshevU hermiteH laguerreL laguerreL legendreP

— package ORTHPOL OrthogonalPolynomialFunctions —

)abbrev package ORTHPOL OrthogonalPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created: 1990
++ Date Last Updated: June 25, 1991
++ Description:
++ This package provides orthogonal polynomials as functions on a ring.

OrthogonalPolynomialFunctions(R: CommutativeRing): Exports == Impl where
NNI ==> NonNegativeInteger
RN ==> Fraction Integer
Exports == with

chebyshevT: (NNI, R) -> R
++ chebyshevT(n,x) is the n-th Chebyshev polynomial of the first
++ kind, \spad{T[n](x)}. These are defined by
++ \spad{(1-t*x)/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}

chebyshevU: (NNI, R) -> R
++ chebyshevU(n,x) is the n-th Chebyshev polynomial of the second
++ kind, \spad{U[n](x)}. These are defined by
++ \spad{1/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}

hermiteH: (NNI, R) -> R
++ hermiteH(n,x) is the n-th Hermite polynomial, \spad{H[n](x)}. 
++ These are defined by
++ \( \exp(2t^2-t^2) = \sum H[n](x) t^n/n! \), \( n = 0 \).

\text{laguerreL: (NNI, R) -> R}
++ \( \text{laguerreL}(n,x) \) is the \( n \)-th Laguerre polynomial, \( L[n](x) \).
++ These are defined by
++ \( \exp(-x/(1-t))/(1-t) = \sum L[n](x) t^n/n! \), \( n = 0 \).

\text{laguerreL: (NNI, NNI, R) -> R}
++ \( \text{laguerreL}(m,n,x) \) is the associated Laguerre polynomial,
++ \( \text{laguerreL}(m\leq n)(x) \). This is the \( m \)-th derivative of \( L[n](x) \).

if \( R \) has Algebra RN then

\text{legendreP: (NNI, R) -> R}
++ \( \text{legendreP}(n,x) \) is the \( n \)-th Legendre polynomial,
++ \( P[n](x) \). These are defined by
++ \( 1/\sqrt{1-2x^2+t^2} = \sum P[n](x) t^n \), \( n = 0 \).

Impl => add
p0, p1: R
cx: Integer

import IntegerCombinatoricFunctions()

\text{laguerreL}(n, x) ==
n = 0 => 1
(p1, p0) := (-x + 1, 1)
for i in 1..n-1 repeat
  (p1, p0) := ((2*i::R + 1 - x)*p1 - i**2*p0, p1)
p1

\text{laguerreL}(m, n, x) ==
ni := n::Integer
mi := m::Integer
cx := (-1)**m * binomial(ni,ni-mi) * factorial(ni)
p0 := 1
p1 := cx::R
for j in 1..ni-mi repeat
  cx := -cx*(ni-mi-j+1)
  cx := (cx exquo ((mi+j)*j))::Integer
p0 := p0 * x
p1 := p1 + cx*p0
p1

\text{chebyshevT}(n, x) ==
n = 0 => 1
(p1, p0) := (x, 1)
for i in 1..n-1 repeat
  (p1, p0) := (2*x*p1 - p0, p1)
p1

\text{chebyshevU}(n, x) ==
n = 0 => 1
\begin{verbatim}
(p1, p0) := (2*x, 1)
for i in 1..n-1 repeat
  (p1, p0) := (2*x*p1 - p0, p1)
p1
hermiteH(n, x) ==
n = 0 => 1
(p1, p0) := (2*x, 1)
for i in 1..n-1 repeat
  (p1, p0) := (2*x*p1 - 2*i*p0, p1)
p1
if R has Algebra RN then
legendreP(n, x) ==
n = 0 => 1
p0 := 1
p1 := x
for i in 1..n-1 repeat
  c: RN := 1/(i+1)
  (p1, p0) := (c*((2*i+1)*x*p1 - i*p0), p1)
p1
\end{verbatim}

package OUT OutputPackage

\begin{verbatim}
)set break resume
)sys rm -f OutputPackage.output
)spool OutputPackage.output
)set message test on
)set message auto off
)clear all
-- S 1 of 1
\end{verbatim}
OutputPackage (OUT)

Exports:
  output  outputList

--- package OUT OutputPackage ---

)abbrev package OUT OutputPackage
++ Author: Stephen M. Watt
++ Date Created: February 1986
++ Date Last Updated: October 27 1995 (MCD)
++ Description:
++ OutPackage allows pretty-printing from programs.

OutPackage: with

output: String -> Void
  ++ output(s) displays the string s on the ‘‘algebra output’’
  ++ stream, as defined by \spadsyscom{set output algebra}.

output: OutputForm -> Void
  ++ output(x) displays the output form x on the
  ++ ‘‘algebra output’’ stream, as defined by
  ++ \spadsyscom{set output algebra}.

output: (String, OutputForm) -> Void
  ++ output(s,x) displays the string s followed by the form x
  ++ on the ‘‘algebra output’’ stream, as defined by
  ++ \spadsyscom{set output algebra}.

outputList: (List Any) -> Void
  ++ outputList(l) displays the concatenated components of the
  ++ list l on the ‘‘algebra output’’ stream, as defined by
  ++ \spadsyscom{set output algebra}; quotes are stripped
  ++ from strings.

== add

--ExpressionPackage()
E ==> OutputForm
putout ==> mathprint$Lisp

s: String
e: OutputForm
l: List Any

output e ==
  mathprint(e)$Lisp
  void()

-- Note that we have to do the pretend here because otherwise we will
-- try to load STRING which is not yet compiled during build.

output s ==
  output(s pretend OutputForm)

output(s,e) ==
  output blankSeparate [s pretend OutputForm, e]

outputList(l) == -- MGR
  output hconcat
[if retractable?(x)$AnyFunctions1(String) then
  message(retract(x)$AnyFunctions1(String))$OutputForm
else
  x::OutputForm
for x in l]

—— OUT.dotabb ——

"OUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OUT"]
"Package" [color="#FF4488"]
"OUT" -> "Package"

——
package PAFF PackageForAlgebraicFunctionField

--- PackageForAlgebraicFunctionField.input ---

)set break resume
)sys rm -f PackageForAlgebraicFunctionField.output
)spool PackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForAlgebraicFunctionField
--R
--R PackageForAlgebraicFunctionField(K: Field, symb: List(Symbol), BLMET: BlowUpMethodCategory) is a package
--R Abbreviation for PackageForAlgebraicFunctionField is PAFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFF
--R
--R------------------------------- Operations --------------------------------
--R fullDesTree : () -> Void fullInfClsPt : () -> Void
--R genus : () -> NonNegativeInteger genusNeg : () -> Integer
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R adjunctionDivisor : () -> Divisor(Places(K))
--R classNumber : () -> Integer if K has FINITE
--R desingTree : () -> List(DesingTree(InfClsPt(K,symb,BLMET))))
--R desingTreeNoFullParam : () -> List(DesingTree(InfClsPt(K,symb,BLMET))))
--R eval : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> K
--R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),Places(K)) -> K
--R eval : (Fraction(DistributedMultivariatePolynomial(symb,K)),Places(K)) -> K
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> Union(K,"failed")
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),Places(K)) -> Union(K,"failed")
--R evalIfCan : (Fraction(DistributedMultivariatePolynomial(symb,K)),Places(K)) -> Union(K,"failed")
--R findOrderOfDivisor : (Divisor(Places(K)),Integer,Integer) -> Record(ord: Integer,num: DistributedMultivariatePolynomial(symb,K),den: DistributedMultivariatePolynomial(symb,K),upTo: Integer)
--R goppaCode : (Divisor(Places(K)),Divisor(Places(K))) -> Matrix(K)
--R goppaCode : (Divisor(Places(K)),List(Places(K))) -> Matrix(K)
--R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariatePolynomial(symb,K)
--R interpolateForms : (Divisor(Places(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
--R interpolateFormsForFact : (Divisor(Places(K)),List(DistributedMultivariatePolynomial(symb,K))) -> List(DistributedMultivariatePolynomial(symb,K))
--R intersectionDivisor : DistributedMultivariatePolynomial(symb,K) -> Divisor(Places(K))
--R 1Basis : (Divisor(Places(K)),NonNegativeInteger) -> List(Fraction(DistributedMultivariatePolynomial(symb,K)))
--R 1Basis : Divisor(Places(K)) -> Record(num: List(DistributedMultivariatePolynomial(symb,K)),den: DistributedMultivariatePolynomial(symb,K))
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R parametrize : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> NeitherSparseOrDensePowerSeries(K)
--R placesAbove : ProjectivePlane(K) -> List(Places(K))
--R placesOfDegree : PositiveInteger -> List(Places(K)) if K has FINITE
--R pointDominateBy : Places(K) -> ProjectivePlane(K)
--R projectivePoint : List(K) -> ProjectivePlane(K)
--R rationalPlaces : () -> List(Places(K))
--R rationalPoints : () -> List(ProjectivePlane(K))
--R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(symb,K)
--R setSingularPoints : List(ProjectivePlane(K)) -> List(ProjectivePlane(K))
--R singularPoints : () -> List(ProjectivePlane(K))
--R theCurve : () -> DistributedMultivariatePolynomial(symb,K)
--R
--E 1
)
}

---

--- PackageForAlgebraicFunctionField.help ---

====================================================================
PackageForAlgebraicFunctionField examples
====================================================================

Part of the PAFF package

See Also:
o )show PackageForAlgebraicFunctionField

---
PackageForAlgebraicFunctionField (PAFF)

Exports:
adjunctionDivisor  classNumber  desingTree
desingTreeWoFullParam  eval  evalIfCan
findOrderOfDivisor  fullDesTree  fullInfClsPt
genus  genusNeg  goppaCode
homogenize  interpolateForms  interpolateFormsForFact
intersectionDivisor  lBasis  LPolynomial
numberOfPlacesOfDegree  numberPlacesDegExtDeg  numberRatPlacesExtDeg
parametrize  placesAbove  placesOfDegree
pointDominateBy  projectivePoint  rationalPlaces
rationalPoints  setCurve  setSingularPoints
singularPoints  theCurve  ZetaFunction

--- package PAFF PackageForAlgebraicFunctionField ---

)abbrev package PAFF PackageForAlgebraicFunctionField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that implements the Brill-Noether algorithm.
++ Part of the PAFF package

PackageForAlgebraicFunctionField(K,symb,BLMET):Exports == Implementation where
K:Field
symb : List(Symbol)
PolyRing ==&gt; DistributedMultivariatePolynomial(symb,K)
E ==&gt; DirectProduct(#symb,NonNegativeInteger)
BLMET : BlowUpMethodCategory

AFP    ==&gt; AffinePlane(K)
ProjPt  ==&gt; ProjectivePlane(K)
PCS    ==&gt; NeitherSparseOrDensePowerSeries(K)
Plc  ==>  Places(K)
DIVISOR  ==>  Divisor(Plc)
InfClsPoint  ==>  InfClsPt(K,symb,BLMET)
DesTree  ==>  DesingTree(InfClsPoint)
FRACPOLY  ==>  Fraction PolyRing
NNI  ==>  NonNegativeInteger
PI  ==>  PositiveInteger
UTSZ  ==>  UnivariateTaylorSeriesCZero(Integer,t)
PAFFPC  ==>  GeneralPackageForAlgebraicFunctionField
PACKPOLY  ==>  PackageForPoly(K,PolyRing,E,#symb)
BP  ==>  PAFFPC(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

Exports  ==>  with
  homogenize: (PolyRing,Integer) -> PolyRing
  interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)
  fullDesTree: () -> Void
  fullInfClsPt: () -> Void
  setCurve: PolyRing -> PolyRing
  projectivePoint: List K -> ProjPt
  pointDominateBy : Plc -> ProjPt
    ++ pointDominateBy(pl) returns the projective point dominated
    ++ by the place pl.
  placesAbove: ProjPt -> List Plc
  setSingularPoints: List ProjPt -> List ProjPt
  goppaCode: (DIVISOR,DIVISOR) -> Matrix K
  goppaCode: (DIVISOR,List(Plc)) -> Matrix K
  rationalPlaces: () -> List Plc
    ++ rationalPlaces returns all the rational places of the
    ++ curve defined by the polynomial given to the package.
  theCurve: () -> PolyRing
    ++ theCurve returns the specified polynomial for the package.
  genus: () -> NNI
    ++ genus returns the genus of the curve defined by the polynomial
    ++ given to the package.
  genusNeg: () -> Integer
desingTreeWoFullParam : () -> List DesTree
++ desingTreeWoFullParam returns the desingularisation trees at all
++ singular points of the curve defined by the polynomial given to
++ the package. The local parametrizations are not computed.

desingTree: () -> List DesTree
++ desingTree returns the desingularisation trees at all singular
++ points of the curve defined by the polynomial given to the package.

rationalPoints: () -> List(ProjPt)
singularPoints: () -> List(ProjPt)
++ rationalPoints() returns the singular points of the curve defined by the polynomial given to the package.
++ If the singular points lie in an extension of the specified ground field an error message is issued specifying the extension degree needed to find all singular points.

parametrize: (PolyRing,Plc) -> PCS
++ parametrize(f,pl) returns a local parametrization of f at the place pl.

lBasis: (DIVISOR,NNI) -> List FRACPOLY
++ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> Record(ord:Integer,num:PolyRing,den:PolyRing)
++ findOrderOfDivisor(d,n) returns a basis of the interpolate forms of degree n of the divisor d.

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
++ interpolateForms(d,n) returns a basis of the interpolate forms of degree n of the divisor d.

eval: (PolyRing,Plc) -> K
++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FRACPOLY,Plc) -> K
++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,pl) evaluate f at the place pl
++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
++ (returns "failed" if it is a pole).
evalIfCan: (FRACPOLY,Plc) -> Union(K,"failed")
++ evalIfCan(u,pl) evaluate the function u at the place pl
++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
++ intersectionDivisor(pol) compute the intersection divisor (the
++ Cartier divisor) of the form pol with the curve. If some
++ intersection points lie in an extension of the ground field,
++ an error message is issued specifying the extension degree
++ needed to find all the intersection points.
++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
++ adjunctionDivisor computes the adjunction divisor of the plane
++ curve given by the polynomial set with the function setCurve.

if K has Finite then --should we say LocallyAlgebraicallyClosedField??

LPolynomial: () -> SparseUnivariatePolynomial Integer
++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

classNumber: () -> Integer
++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
++ placesOfDegree(d) returns all places of degree d of the curve.

numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
++ places in the constant field extension of degree n

numberPlacesDegExtDeg: (PI, PI) -> Integer
++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of degree n

ZetaFunction: () -> UTSZ
++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial
Implementation ==> add

import BP

homogenize(pol,n) == homogenize(pol,n)$PACKPOLY

pointDominateBy(pl) == pointDominateBy(pl)$BP

placesAbove(pt) == placesAbove(pt)$BP

setSingularPoints(lspt) == setSingularPoints(lspt)$BP

projectivePoint(lpt) == projectivePoint(lpt)$ProjPt

interpolateFormsForFact(d,lm) ==
  interpolateFormsForFact(d,lm)$BP

if K has Finite then
  goppaCode(d:DIVISOR,lp:List(Plc)) ==
    lb := lBasis(d)
    dd := lb.den
    ll := [[eval(f, dd, pl) for pl in lp] for f in lb.num]
    matrix ll

  goppaCode(d:DIVISOR, p:DIVISOR) ==
    lp := supp p
    goppaCode(d, lp)

ZetaFunction == ZetaFunction()$BP

ZetaFunction(d) == ZetaFunction(d)$BP

numberOfPlacesOfDegree(i) == numberOfPlacesOfDegree(i)$BP

placesOfDegree(i) == placesOfDegree(i)$BP

numberRatPlacesExtDeg(extDegree) == numberRatPlacesExtDeg(extDegree)$BP

numberPlacesDegExtDeg(degree, extDegree) ==
  numberPlacesDegExtDeg(degree, extDegree)$BP

LPolynomial == LPolynomial()$BP

LPolynomial(extDeg) == LPolynomial(extDeg)$BP

classNumber == classNumber()$BP

rationalPlaces == rationalPlaces()$BP

rationalPoints == rationalPoints()$BP
crvLocal:PolyRing

eval(f:PolyRing,pl:Plc)==
  dd:= degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  eval(f,pl)$BP

evalIfCan(f:PolyRing,pl:Plc)==
  dd:= degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  evalIfCan(f,pl)$BP

setCurve(pol)==setCurve(pol)$BP

lBasis(divis)==lBasis(divis)$BP

genus==genus()$BP

genusNeg==genusNeg()$BP

theCurve==theCurve()$BP

desingTree==desingTree()$BP

desingTreeWoFullParam== desingTreeWoFullParam()$BP

-- compute the adjunction divisor of the curve using
-- adjunctionDivisor from DesingTreePackage
adjunctionDivisor == adjunctionDivisor()$BP

singularPoints==singularPoints()$BP

parametrize(f,pl)==parametrize(f,pl)$BP

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==interpolateForms(d,n)$BP

eval(f:PolyRing,g:PolyRing,pl:Plc)==eval(f,g,pl)$BP

eval(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
  eval(ff,gg,pl)

evalIfCan(f:PolyRing,g:PolyRing,pl:Plc)==evalIfCan(f,g,pl)$BP

evalIfCan(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField

-- PackageForAlgebraicFunctionFieldOverFiniteField.input --

)set break resume
)sys rm -f PackageForAlgebraicFunctionFieldOverFiniteField.output
)spool PackageForAlgebraicFunctionFieldOverFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForAlgebraicFunctionFieldOverFiniteField
--R
--R PackageForAlgebraicFunctionFieldOverFiniteField(K: FiniteFieldCategory,symb: List(Symbol),BLMET: BlowUpMethodCategory) is a package constructor
--R Abbreviation for PackageForAlgebraicFunctionFieldOverFiniteField is PAFFFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFFFF
CHAPTER 17. CHAPTER P

--R
--R------------------------------------------------------------- Operations ----------------------------------
--R fullDesTree : () -> Void fullInfClsPt : () -> Void
--R genus : () -> NonNegativeInteger genusNeg : () -> Integer
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R adjunctionDivisor : () -> Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
--R classNumber : () -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R desingTree : () -> List(DesingTree(InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)))
--R desingTreeWoFullParam : () -> List(DesingTree(InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)))
--R eval : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
--R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
--R eval : (Fraction(DistributedMultivariatePolynomial(symb,K)),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
--R evalIfCan : (Fraction(DistributedMultivariatePolynomial(symb,K)),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
--R findOrderOfDivisor : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Integer,Integer) -> Record(ord: Integer,num: DistributedMultivariatePolynomial(symb,K),den: DistributedMultivariatePolynomial(symb,K),upTo: Integer)
--R goppaCode : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Matrix(K)
--R goppaCode : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Matrix(K)
--R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariatePolynomial(symb,K)
--R interpolateForms : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
--R interpolateFormsForFact : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),List(DistributedMultivariatePolynomial(symb,K))) -> List(DistributedMultivariatePolynomial(symb,PseudoAlgebraicClosureOfFiniteField(K)))
--R intersectionDivisor : DistributedMultivariatePolynomial(symb,K) -> Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
--R lBasis : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
--R lBasis : Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Record(num: List(DistributedMultivariatePolynomial(symb,K)),den: DistributedMultivariatePolynomial(symb,K))
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R parametrize : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
--R placesAbove : ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K) -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
--R placesOfDegree : PositiveInteger -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
--R pointDominateBy : PlacesOverPseudoAlgebraicClosureOfFiniteField(K) -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
--R projectivePoint : List(DistributedMultivariatePolynomial(symb,K)) -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
--R rationalPlaces : () -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
--R rationalPoints : () -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
--R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(symb,K)
--R setSingularPoints : List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)) -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
--R singularPoints : () -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
--R theCurve : () -> DistributedMultivariatePolynomial(symb,K)
--R translateToOrigin : (DistributedMultivariatePolynomial(symb,K),ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
--R
--E 1

)spool
)lisp (bye)
PACKAGE PAFFFF PACKAGE FOR ALGEBRAIC FUNCTION FIELD OVER FINITE FIELD

<table>
<thead>
<tr>
<th>PackageForAlgebraicFunctionFieldOverFiniteField</th>
<th>help</th>
</tr>
</thead>
</table>

====================================================================
PackageForAlgebraicFunctionFieldOverFiniteField examples
====================================================================

Part of the PAFF package

See Also:
o )show PackageForAlgebraicFunctionFieldOverFiniteField

PackageForAlgebraicFunctionFieldOverFiniteField (PAFFFF)

---

adjunctionDivisor
desingTreeWoFullParam
findOrderOfDivisor
genus
homogenize
intersectionDivisor
numberOfPlacesOfDegree
parametrize
pointDominateBy
rationalPoints
singularPoints
ZetaFunction
classNumber
desingTree
eval
evalIfCan
fullDesTree
fullInfClsPt
genusNeg
interpolateForms
interpolateFormsForFact
lBasis
LPolynomial
numberPlacesDegExtDeg
numberRatPlacesExtDeg
placesAbove
placesOfDegree
projectivePoint
rationalPlaces
setCurve
setSingularPoints
theCurve
translateToOrigin

---

package PAFFFF PackageForAlgebraicFunctionFieldOverFinite-Field
PackageForAlgebraicFunctionFieldOverFiniteField(K,symb,BLMET):Exp == Impl where

K:FiniteFieldCategory -- Field
symb : List(Symbol)
BLMET : BlowUpMethodCategory

DK ==> PseudoAlgebraicClosureOfFiniteField(K)
PolyRing ==> DistributedMultivariatePolynomial(symb,K)
PolyRing2 ==> DistributedMultivariatePolynomial(symb,DK)
ProjPt ==> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
UTSZ ==> UnivariateTaylorSeriesCZero(Integer,t)
PAFFPC ==> GeneralPackageForAlgebraicFunctionField
PCS ==> NeitherSparseOrDensePowerSeries(DK)
Plc ==> PlacesOverPseudoAlgebraicClosureOfFiniteField(K)
DIVISOR ==> Divisor(Plc)
InfClsPoint ==> InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)
DesTree ==> DesingTree(InfClsPoint)
FracPoly ==> Fraction PolyRing
PackPoly ==> PackageForPoly(DK,PolyRing2,E,#symb)
E ==> DirectProduct(#symb,NNI)
BP ==> PAFFPC(DK,symb,PolyRing2,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

Exp ==> with
  homogenize: (PolyRing,Integer) -> PolyRing

fullDesTree: () -> Void
fullInfClsPt: () -> Void

setCurve: PolyRing -> PolyRing
translateToOrigin: (PolyRing, ProjPt) -> PolyRing2

goppaCode: (DIVISOR,DIVISOR) -> Matrix K

goppaCode: (DIVISOR,List(Plc)) -> Matrix K

pointDominateBy : Plc -> ProjPt
  ++ pointDominateBy(pl) returns the projective point dominated
  ++ by the place pl.
placesAbove: ProjPt -> List Plc
projectivePoint: List DK -> ProjPt
setSingularPoints: List ProjPt -> List ProjPt
rationalPlaces: () -> List Plc
++ rationalPlaces returns all the rational places of the curve defined by the polynomial given to the package.
theCurve: () -> PolyRing
++ theCurve returns the specified polynomial for the package.
genus: () -> NNI
++ genus returns the genus of the curve defined by the polynomial given to the package.
genusNeg: () -> Integer
designTreeWoFullParam : () -> List DesTree
++ designTreeWoFullParam returns the desingularisation trees at all singular points of the curve defined by the polynomial given to the package. The local parametrizations are not computed.
designTree: () -> List DesTree
++ designTree returns the desingularisation trees at all singular points of the curve defined by the polynomial given to the package.
rationalPoints: () -> List(ProjPt)
singularPoints: () -> List(ProjPt)
++ rationalPoints() returns the singular points of the curve defined by the polynomial given to the package.
++ If the singular points lie in an extension of the specified ground field an error message is issued specifying the extension degree needed to find all singular points.
parametrize: (PolyRing,Plc) -> PCS
++ parametrize(f,pl) returns a local parametrization of f at the place pl.
lBasis: (DIVISOR,NNI) -> List FracPoly
++ lBasis computes a basis associated to the specified divisor
findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _
++ findOrderOfDivisor returns the order of the divisor
interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing2)
interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
++ interpolateForms(d,n) returns a basis of the interpolate forms of
++ degree n of the divisor d.

eval: (PolyRing,Plc) -> K
++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FracPoly,Plc) -> K
++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,pl) evaluate f at the place pl
++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
++ (returns "failed" if it is a pole).

evalIfCan: (FracPoly,Plc) -> Union(K,"failed")
++ evalIfCan(u,pl) evaluate the function u at the place pl
++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
++ intersectionDivisor(pol) compute the intersection divisor of the
++ form pol with the curve.
++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
++ adjunctionDivisor computes the adjunction divisor of the plane
++ curve given by the polynomial defined by setCurve.

if DK has Finite then --should we say LocallyAlgebraicallyClosedField??

LPolynomial: () -> SparseUnivariatePolynomial Integer
++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

classNumber: () -> Integer
++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
++ placesOfDegree(d) returns all places of degree d of the
++ curve.
PACKAGE PAFF PACKAGE FORMALGEBRAICFUNCTIONFIELD OVER FINITE FIELD

numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
++ places in the constant field extension of degree n

numberPlacesDegExtDeg: (PI, PI) -> Integer
++ numberPlacesDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of
++ degree n

ZetaFunction: () -> UTSZ
++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial

Impl ==> add
import BP

homogenize(pol,n) == homogenize(pol,n)$PackageForPoly(K,PolyRing,E,#symb)
toPolyRing2: PolyRing -> PolyRing2
toPolyRing: PolyRing2 -> PolyRing

projectivePoint(lpt)==projectivePoint(lpt)$ProjPt
pointDominateBy(pl)== pointDominateBy(pl)$BP
placesAbove(pt)== placesAbove(pt)$BP
setSingularPoints(lspt)== setSingularPoints(lspt)$BP

findOrderOfDivisor(divis,lb,hb) ==
  ens:=findOrderOfDivisor(divis,lb,hb)$BP
  [ens.ord, toPolyRing ens.num, toPolyRing ens.den, ens.upTo]

setCurve(pol)==
  ooo:=setCurve(toPolyRing2 pol)$BP
  pol

ZetaFunction == ZetaFunction()$BP
ZetaFunction(d) == ZetaFunction(d)$BP

numberOfPlacesOfDegree(i)==numberOfPlacesOfDegree(i)$BP
placesOfDegree(i) == placesOfDegree(i) \BP

numberRatPlacesExtDeg(extDegree) == numberRatPlacesExtDeg(extDegree) \BP

numberPlacesDegExtDeg(degree, extDegree) ==
  numberPlacesDegExtDeg(degree, extDegree) \BP

LPolynomial == LPolynomial() \BP

LPolynomial(extDeg) == LPolynomial(extDeg) \BP

classNumber == classNumber() \BP

rationalPlaces == rationalPlaces() \BP

rationalPoints == rationalPoints() \BP

goppaCode(d: DIVISOR, lp: List(Plc)) ==
  lb := lBasis(d)
  dd := lb.den
  ll := [[eval(f, dd, pl) for pl in lp] for f in lb.num]
  matrix ll

goppaCode(d: DIVISOR, p: DIVISOR) ==
  lp := supp p
goppaCode(d, lp)

toPolyRing(pol) ==
  zero?(pol) => 0 \PolyRing
  lc := leadingCoefficient pol
  lce: K := retract lc
  lm := leadingMonomial pol
  lt := degree lm
  monomial(lce, lt) \PolyRing + toPolyRing( reductum pol )

toPolyRing2(pol) ==
  zero?(pol) => 0 \PolyRing2
  lc := leadingCoefficient pol
  lce: DK := lc :: DK
  lm := leadingMonomial pol
  lt := degree lm
  monomial(lce, lt) \PolyRing2 + toPolyRing2( reductum pol )

evalIfCan(f: PolyRing, pl: Plc) ==
  dd := degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  ee := evalIfCan(toPolyRing2 f, pl) \BP
  ee case "failed" => "failed"
  retract ee
eval(f:PolyRing,p1:P1c)==
   dd:= degree p1
   "one?(dd) => error " cannot evaluate at place of degree greater than one"
   ee:=eval(toPolyRing2 f,p1)$BP
   retract ee

lBasis(divis)==
   ans:=lBasis(divis)$BP
   nn:=ans.num
   dd:=ans.den
   nnd:=[toPolyRing pol for pol in nn]
   ddd:=toPolyRing dd
   [nnd,ddd]

genus==genus()$BP

genusNeg==genusNeg()$BP

theCurve==
   ccc:= theCurve()$BP
   toPolyRing ccc

desingTree==desingTree()$BP

desingTreeWoFullParam== desingTreeWoFullParam()$BP

-- compute the adjunction divisor of the curve using
-- adjunctionDivisor from DesingTreePackage
adjunctionDivisor == adjunctionDivisor()$BP

singularPoints==singularPoints()$BP

parametrize(f,p1)==
   ff:= toPolyRing2 f
   parametrize(ff,p1)$BP

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==
   ans:=interpolateForms(d,n)$BP
   [toPolyRing pol for pol in ans]

interpolateFormsForFact(d,lm)==
   lm2:List PolyRing2 := [ toPolyRing2 p for p in lm]
   interpolateFormsForFact(d,lm2)$BP

evalIfCan(ff:PolyRing,gg:PolyRing,p1:P1c)==
   dd:= degree p1
   "one?(dd) => error " cannot evaluate at place of degree greater than one"
   f:=toPolyRing2 ff
\begin{verbatim}
g:=toPolyRing2 gg
eq:=evalIfCan(f,g,pl)$BP
eq case "failed" => "failed"
retract \( \text{ee} \)

eval(ff:PolyRing,gg:PolyRing,pl:Plc)==
  dd:= degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  f:=toPolyRing2 ff
g:=toPolyRing2 gg
  ee:=eval(f,g,pl)$BP
  retract \( \text{ee} \)

evalIfCan(u:FracPoly,pl:Plc)==
  ff:=numer u
gg:=denom u
  evalIfCan(ff,gg,pl)

eval(u:FracPoly,pl:Plc)==
  ff:=numer u
gg:=denom u
  eval(ff,gg,pl)

intersectionDivisor(pol)==
  polu:=toPolyRing2 pol
  intersectionDivisor(polu)$BP

fullDesTree==
  fullOutput()$DesTree => fullOutput(false())$DesTree
  fullOutput(true())$DesTree

fullInfClsPt==
  fullOutput()$InfClsPoint => fullOutput(false())$InfClsPoint
  fullOutput(true())$InfClsPoint
\end{verbatim}
package PFORP PackageForPoly

--- PackageForPoly.input ---

)set break resume
)sys rm -f PackageForPoly.output
)spool PackageForPoly.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForPoly

--R
--R PackageForPoly(R: Ring, PolyRing: FiniteAbelianMonoidRing(R,E), E: DirectProductCategory(dim, NonNegativeInteger)) is a package constructor
--R Abbreviation for PackageForPoly is PFORP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFORP
--R
--R------------------------------- Operations --------------------------------
--R constant : PolyRing -> R
--R firstExponent : PolyRing -> E
--R minimalForm : PolyRing -> PolyRing
--R degOneCoef : (PolyRing, PositiveInteger) -> R
--R degree : (PolyRing, Integer) -> NonNegativeInteger
--R degreeOfMinimalForm : PolyRing -> NonNegativeInteger
--R homogenize : (PolyRing, Integer) -> PolyRing
--R listAllMono : NonNegativeInteger -> List(PolyRing)
--R listAllMonoExp : Integer -> List(E)
--R listVariable : () -> List(PolyRing)
--R mapExponents : ((E -> E), PolyRing) -> PolyRing
--R monomials : PolyRing -> List(PolyRing)
--R replaceVarByOne : (PolyRing, Integer) -> PolyRing
--R replaceVarByZero : (PolyRing, Integer) -> PolyRing
--R subs1stVar : (PolyRing, PolyRing) -> PolyRing
--R subs2ndVar : (PolyRing, PolyRing, PolyRing) -> PolyRing
--R subsInVar : (PolyRing, PolyRing, Integer) -> PolyRing
--R totalDegree : PolyRing -> NonNegativeInteger
--R translate : (PolyRing, List(R), Integer) -> PolyRing
--R translate : (PolyRing, List(R)) -> PolyRing
--R univariate : PolyRing -> SparseUnivariatePolynomial(R)
--R
--E 1

)spool
)lisp (bye)
--- PackageForPoly.help ---

====================================================================

PackageForPoly examples
====================================================================

The following is part of the PAFF package

See Also:

o )show PackageForPoly

PackageForPoly (PFORP)

--- package PFORP PackageForPoly ---

)abbrev package PFORP PackageForPoly
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PackageForPoly(R, PolyRing, E, dim): public == private where
  R: Ring -- was Field but change for SolveTree package. 21/01/98
  dim: NonNegativeInteger
  E: DirectProductCategory(dim, NonNegativeInteger)
  PolyRing: FiniteAbelianMonoidRing(R, E)

Term => Record(k:E, c:R)
PI => PositiveInteger
NNI => NonNegativeInteger
INT => Integer

public == with
  mapExponents: (E->E, PolyRing) -> PolyRing
  degree: (PolyRing, Integer) -> NNI
  univariate: PolyRing -> SparseUnivariatePolynomial(R)
  totalDegree: PolyRing -> NNI
  subs1stVar: (PolyRing, PolyRing) -> PolyRing
  subs2ndVar: (PolyRing, PolyRing) -> PolyRing
  subsInVar: (PolyRing, PolyRing, Integer) -> PolyRing
  minimalForm: PolyRing -> PolyRing
    ++ minimalForm(pol) returns the minimal forms of the polynomial pol.
  firstExponent: PolyRing -> E
    ++ firstExponent(pol) returns the exponent of the first term in the
    ++ representation of pol. Not to be confused with the leadingExponent
    ++ which is the highest exponent according to the order
    ++ over the monomial.
  replaceVarByZero: (PolyRing, Integer) -> PolyRing
    ++ replaceVarByZero(pol, a) evaluate to zero the variable in pol
    ++ specified by the integer a.
  replaceVarByOne: (PolyRing, Integer) -> PolyRing
    ++ replaceVarByOne(pol, a) evaluate to one the variable in pol
    ++ specified by the integer a.
  translate: (PolyRing, List R, Integer) -> PolyRing
    ++ translate(pol, [a, b, c], 3) apply to pol the
    ++ linear change of coordinates, x->x+a, y->y+b, z->1.
  translate: (PolyRing, List R) -> PolyRing
    ++ translate(pol, [a, b, c]) apply to pol the
    ++ linear change of coordinates, x->x+a, y->y+b, z->z+c
degOneCoef: (PolyRing,PI) -> R
  ++ degOneCoef(pol,n) returns the coefficient in front of the monomial
  ++ specified by the positive integer.

constant: PolyRing -> R
  ++ constant(pol) returns the constant term of the polynomial.

homogenize: (PolyRing,INT) -> PolyRing
  ++ homogenize(pol,n) returns the homogenized polynomial of pol
  ++ with respect to the n-th variable.

listAllMonoExp: Integer -> List E
  ++ listAllMonoExp(l) returns all the exponents of degree l

listAllMono: NNI -> List PolyRing
  ++ listAllMono(l) returns all the monomials of degree l

degreeOfMinimalForm: PolyRing -> NNI
  ++ degreeOfMinimalForm does what it says

listVariable: () -> List PolyRing

monomials: PolyRing -> List PolyRing

private == add
  import PolyRing

  monomials(pol)==
    zero? pol => empty()
    lt:=leadingMonomial pol
    cons( lt , monomials reductum pol )

  lll: Integer -> E
  lll(i) ==
    le:=new( dim , 0$NNI)$List(NNI)
    le.i := 1
    directProduct( vector(le)$Vector(NNI) )$E

listVariable==
  [monomial(1,ee)$PolyRing for ee in [lll(i) for i in 1..dim]]

univariate(pol)==
  zero? pol => 0
  d:=degree pol
  lc:=leadingCoefficient pol
  td := reduce("+", entries d)
  monomial(lc,td)$SparseUnivariatePolynomial(R)+univariate(reductum pol)
collectExpon: List Term -> PolyRing
translateLocal: (PolyRing,List R,Integer) -> PolyRing
lA: (Integer,Integer) -> List List NNI
toListRep: PolyRing -> List Term
exponentEntryToZero: (E,Integer) -> E
exponentEntryZero?: (E,Integer) -> Boolean
homogenizeExp: (E,NNI,INT) -> E
translateMonomial: (PolyRing,List R,INT,R) -> PolyRing

leadingTerm: PolyRing -> Term
mapExponents(f,pol)==
  zero?(pol) => 0
  lt:=leadingTerm pol
  newExp:= f(lt.k)
  newMono:= monomial(lt.c,newExp)$PolyRing
  newMono + mapExponents(f,reductum pol)

collectExpon(pol)==
  empty? pol => 0
  ft:=first pol
  monomial(ft.c,ft.k) + collectExpon( rest pol )

subs1stVar(pol, spol)==
  zero? pol => 0
  lexpE:= degree pol
  lexp:List NNI:= parts lexpE
  coef:= leadingCoefficient pol
  coef * spol ** lexp.1 * second(listVariable())**lexp.2 _
    + subs1stVar( reductum pol, spol )

subs2ndVar(pol, spol)==
  zero? pol => 0
  lexpE:= degree pol
  lexp:List NNI:= parts lexpE
  coef:= leadingCoefficient pol
  coef * first(listVariable())**lexp.1 * spol ** lexp.2 _
    + subs2ndVar( reductum pol, spol )

subsInVar( pol, spol, n)==
  one?( n ) => subs1stVar( pol, spol )
  subs2ndVar(pol,spol)
translate(pol,lpt)==
  zero? pol => 0
  lexp:E:= degree pol
  lexp:List NNI:= parts lexpE
  coef:= leadingCoefficient pol
  trVar:=[(listVariable().i + (lpt.i)::PolyRing)**lexp.i for i in 1..dim]
  coef * reduce("*",trVar,1) + translate(reductum pol , lpt)

translate(poll,lpt,nV)==
  pol:=replaceVarByOne(poll,nV)
  translateLocal(pol,lpt,nV)

translateLocal(pol,lpt,nV)==
  zero?(pol) => 0
  lll:List R:= [l for l in lpt | ^zero?(l)]
  nbOfNonZero:= # lll
  ltk:=leadingMonomial pol
  ltc:=leadingCoefficient pol
  if one?(nbOfNonZero) then
    pol
  else
    translateMonomial(ltk,lpt,nV,ltc) + _
    translateLocal(reductum(pol),lpt,nV)

exponentEntryToZero(exp,nV)==
  pexp:= parts exp
  pexp(nV):=0
  directProduct(vector(pexp)$Vector(NonNegativeInteger))

exponentEntryZero?(exp,nV)==
  pexp:= parts exp
  zero?(pexp(nV))

replaceVarByZero(pol,nV)==
  -- surement le collectExpon ici n'est pas necessaire !!!!
  zero?(pol) => 0
  lRep:= toListRep pol
  reduce("+",_
    [monomial(p.c,p.k)$PolyRing _
    for p in lRep | exponentEntryZero?(p.k,nV) ],0)

replaceVarByOne(pol,nV)==
  zero?(pol) => 0
  lRep:= toListRep pol
  reduce("+",_
    [monomial(p.c,exponentEntryToZero(p.k,nV)$PolyRing for p in lRep],0)

homogenizeExp(exp,deg,nV)==
  lv:List NNI:=parts(exp)
  lv.nV:=(deg+lv.nV - reduce("+",lv)) pretend NNI
directProduct(vector(lv)$Vector(NNI))$E

listTerm: PolyRing -> List E
listTerm(pol)==
  zero? pol => empty
  cons( degree pol, listTerm reductum pol )

degree( a : PolyRing , n : Integer )==
  zero? a => error "Degree for 0 is not defined for this degree fnc"
  "max" / [ ee.n for ee in listTerm a ]

totalDegree p ==
  zero? p => 0
  "max"/[reduce("+", t:(Vector NNI), 0) for t in listTerm p]

homogenize(pol,nV)==
  degP:=totalDegree(pol)
  mapExponents(homogenizeExp(#1,degP,nV),pol)

degOneCoef(p:PolyRing,i:PI)==
  vv:=new(dim,0)$Vector(NNI)
  vv.i:=i
  pd:=directProduct(vv)$E
  lp:=toListRep p
  lc:=[t.c for t in lp | t.k=pd]
  reduce("+",lc,0)

constant(p)==
  vv:=new(dim,0)$Vector(NNI)
  pd:=directProduct(vv)$E
  lp:=toListRep p
  lc:=[t.c for t in lp | t.k=pd]
  reduce("+",lc,0)

degreeOfMinimalForm(pol)==
  totalDegree minimalForm pol

minimalForm(pol)==
  zero?(pol) => pol
  lpol:=toListRep pol
  actTerm:Term:= first lpol
  minDeg:NNI:=reduce("+", parts(actTerm.k))
  actDeg:NNI
  lminForm:List(Term):= [actTerm]
  for p in rest(lpol) repeat
    actDeg:= reduce("+", parts(p.k))
    if actDeg = minDeg then
      lminForm := concat(lminForm,p)
    if actDeg < minDeg then
      minDeg:=actDeg
\begin{verbatim}
lminForm:=[p]
collectExpon lminForm

-- le code de collectExponSort a ete emprunte a D. Augot.

leadingTerm(pol)==
  zero?(pol) => error "no leading term for 0 (message from package)"
lcoef:R:=leadingCoefficient(pol)$PolyRing
lterm:PolyRing:=leadingMonomial(pol)$PolyRing
tt:E:=degree(lterm)$PolyRing
[tt,lcoef]$Term

toListRep(pol)==
  zero?(pol) => empty()
lt:=leadingTerm pol
cons(lt, toListRep reductum pol)

lA(n,l)==
  zero?(n) => [new((l pretend NNI),0)$List(NNI)]
one?(l) => [[(n pretend NNI)]]
concat [[ concat([i],lll) for lll in lA(n-i,l-1) ] for i in 0..n]

listAllMonoExp(l)==
  lst:=lA(l,(dim pretend Integer))
  [directProduct(vector(pexp)$Vector(NNI)) for pexp in lst]

translateMonomial(mono,pt,nV,coef)==
  lexpE:E:= degree mono
  lexp:List NNI:= parts lexpE
  lexp(nV):=0
  trVar:=[(listVariable().i + (pt.i)::PolyRing)** lexp.i for i in 1..dim]
  coef * reduce("*",trVar,1)

listAllMono(l)==
  [monomial(1,e)$PolyRing for e in listAllMonoExp(l)]
\end{verbatim}
package PADEPAC PadeApproximantPackage

--- PadeApproximantPackage.input ---

)set break resume
)sys rm -f PadeApproximantPackage.output
)spool PadeApproximantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PadeApproximantPackage
--E 1

)spool
)lisp (bye)

---

--- PadeApproximantPackage.help ---

====================================================================
PadeApproximantPackage examples
====================================================================

This package computes reliable Padé approximants using a generalized
Viskovatov continued fraction algorithm.

See Also:
o )show PadeApproximantPackage

---
PadeApproximantPackage (PADEPAC)

Exports:

\[ \text{pade} \]

\[
\text{pade}(\text{nd}, \text{dd}, \text{ns}, \text{ds}) \text{ computes the approximant as a quotient of polynomials if it exists for arguments (numerator degree of approximant), (denominator degree of approximant), (numerator series of function), and (denominator series of function).} \\
\text{pade}(\text{nd}, \text{dd}, \text{s}) \text{ computes the quotient of polynomials if it exists with numerator degree at} \\
\]

\[
\text{pade}(\text{nd}, \text{dd}, \text{ns}, \text{ds}) \rightarrow \text{Union(QF, "failed")} \\
\]

\[
\text{pade}(\text{nd}, \text{dd}, \text{s}) \rightarrow \text{Union(QF, "failed")} \\
\]

\[
\text{pade}(\text{nd}, \text{dd}, \text{ns}, \text{ds}) \rightarrow \text{Union(QF, "failed")} \\
\]

\[
\text{pade}(\text{nd}, \text{dd}, \text{s}) \rightarrow \text{Union(QF, "failed")} \\
\]
++ most nd and denominator degree at most dd
++ which matches the series s to order \text{nd + dd}.

Implementation => add
n,m : NNI
u,v : PS
pa := PadeApproximants(R,PS,UP)
pade(n,m,u,v) ==
  ans:=pade(n,m,u,v)$pa
  ans case "failed" => ans
  pt = 0 => ans
  num := numer(ans::QF)
  den := denom(ans::QF)
  xpt : UP := monomial(1,1)-monomial(pt,0)
  num := num(xpt)
  den := den(xpt)
  num/den
pade(n,m,u) == pade(n,m,u,1)

— PADEPAC.dotabb —

"PADEPAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADEPAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PADEPAC" -> "PFECAT"

— PadeApproximants.input —

)set break resume
)sys rm -f PadeApproximants.output
)spool PadeApproximants.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PadeApproximants
--R
--R PadeApproximants(R: Field,PS: UnivariateTaylorSeriesCategory(R),UP: UnivariatePolynomialCategory(R)) in
--R Abbreviation for PadeApproximants is PADE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PADE
--R
--R------------------------------- Operations --------------------------------
--R pade : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(Fraction(UP),"failed")
--R padecf : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(ContinuedFraction(UP),"failed")
--R
--E 1
)

)spool
)lisp (bye)

——
— PadeApproximants.help —
——

====================================================================
PadeApproximants examples
====================================================================

See Also:
o )show PadeApproximants

——

PadeApproximants (PADE)

Exports:
pade padecf

— package PADE PadeApproximants —
abbrev package PADE PadeApproximants
++ Authors: Burge, Hassner & Watt.
++ Date Created: April 1987
++ Date Last Updated: 12 April 1990
++ References:
++ George A. Baker and Peter Graves-Morris
++ ‘Pade Approximants’
++ Description:
++ This package computes reliable Padé approximants using
++ a generalized Viskovatov continued fraction algorithm.

PadeApproximants(R,PS,UP): Exports == Implementation where
  R: Field -- IntegralDomain
  PS: UnivariateTaylorSeriesCategory R
  UP: UnivariatePolynomialCategory R

  NNI ==> NonNegativeInteger
  QF ==> Fraction UP
  CF ==> ContinuedFraction UP

Exports == with
  pade: (NNI,NNI,PS,PS) -> Union(QF,"failed")
    ++ pade(nd,dd,ns,ds)
    ++ computes the approximant as a quotient of polynomials
    ++ (if it exists) for arguments
    ++ nd (numerator degree of approximant),
    ++ dd (denominator degree of approximant),
    ++ ns (numerator series of function), and
    ++ ds (denominator series of function).
  padecf: (NNI,NNI,PS,PS) -> Union(CF,"failed")
    ++ padecf(nd,dd,ns,ds)
    ++ computes the approximant as a continued fraction of
    ++ polynomials (if it exists) for arguments
    ++ nd (numerator degree of approximant),
    ++ dd (denominator degree of approximant),
    ++ ns (numerator series of function), and
    ++ ds (denominator series of function).

Implementation ==> add
  -- The approximant is represented as
  --  p0 + x**a1/(p1 + x**a2/(...))

  PadeRep ==> Record(ais: List UP, degs: List NNI) -- #ais= #degs
  PadeU  ==> Union(PadeRep,"failed") -- #ais= #degs+1

  constInner(up:UP):PadeU == [[up], []]

  truncPoly(p:UP,n:NNI):UP ==
    while n < degree p repeat p := reductum p
truncSeries(s: PS, n: NNI): UP ==
  p: UP := 0
  for i in 0..n repeat p := p + monomial(coefficient(s, i), i)
p

-- Assumes s starts with a<n>x**n + ... and divides out x**n.
divOutDegree(s: PS, n: NNI): PS ==
  for i in 1..n repeat s := quoByVar s
  s

padeNormalize: (NNI, NNI, PS, PS) -> PadeU
padeInner: (NNI, NNI, PS, PS) -> PadeU

pade(l, m, gps, dps) ==
  (ad := padeNormalize(l, m, gps, dps)) case "failed" => "failed"
  plist := ad.ais; dlist := ad.degs
  approx := first(plist) :: QF
  for d in dlist for p in rest plist repeat
    approx := p::QF + (monomial(1, d)$UP :: QF)/approx
  approx

padecf(l, m, gps, dps) ==
  (ad := padeNormalize(l, m, gps, dps)) case "failed" => "failed"
  alist := reverse(ad.ais)
  blist := [monomial(1, d)$UP for d in reverse ad.degs]
  continuedFraction(first(alist),
    blist::Stream UP, (rest alist) :: Stream UP)

padeNormalize(l, m, gps, dps) ==
  zero? dps => "failed"
  zero? gps => constInner 0
  -- Normalize so numerator or denominator has constant term.
  ldeg := min(order dps, order gps)
  if ldeg > 0 then
    dps := divOutDegree(dps, ldeg)
    gps := divOutDegree(gps, ldeg)
  padeInner(l, m, gps, dps)

padeInner(l, m, gps, dps) ==
  zero? coefficient(gps, 0) and zero? coefficient(dps, 0) =>
  error "Pade' problem not normalized."
  plist: List UP := nil()
  alist: List NNI := nil()
  -- Ensure denom has constant term.
  if zero? coefficient(dps, 0) then
    -- g/d = 0 + z**0/(d/g)
    (gps, dps) := (dps, gps)
    (l, m) := (m, l)
  ...
plist := concat(0,plist)
alist := concat(0,alist)

-- Ensure l >= m, maintaining coef(dps,0)^=0.
if l < m then
  -- (a<n>*x**n + a<n+1>*x**n+1 + ...)/b
  -- = x**n/b + (a<n> + a<n+1>*x + ...)/b
  alpha := order gps
  if alpha > l then return "failed"
  gps := divOutDegree(gps, alpha)
  (l,m) := (m,(l-alpha) :: NNI)
  (gps,dps) := (dps,gps)
  plist := concat(0,plist)
  alist := concat(alpha,alist)
  degbd: NNI := l + m + 1
  g := truncSeries(gps,degbd)
  d := truncSeries(dps,degbd)
for j in 0.. repeat
  -- Normalize d so constant coefcels cancel. (B&G-M is wrong)
  d0 := coefficient(d,0)
  d := d0 * d; g := (1/d0) * g
  p : UP := 0; s := g
  if l-m+1 < 0 then error "Internal pade error"
  degbd := (l-m+1) :: NNI
  for k in 1..degbd repeat
    pk := coefficient(s,0)
    p := p + monomial(pk,(k-1) :: NNI)
    s := s - pk*d
    s := s exquo monomial(1,1) :: UP
  plist := concat(p,plist)
  s := 0 => return [plist,alist]
  alpha := minimumDegree(s) + degbd
  alpha > l + m => return [plist,alist]
  alpha > l => return "failed"
  alist := concat(alpha,alist)
  h := (s exquo monomial(1,minimumDegree s)) :: UP
  degbd := (1 + m - alpha) :: NNI
  g := truncPoly(d,degbd)
  d := truncPoly(h,degbd)
  (l,m) := (m,(l-alpha) :: NNI)

---

"PADE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADE"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"PADE" -> "UTSCAT"
package PWFFINTB PAdicWildFunctionFieldIntegralBasis

---

PAdicWildFunctionFieldIntegralBasis.input ---

)set break resume
)sys rm -f PAdicWildFunctionFieldIntegralBasis.output
)spool PAdicWildFunctionFieldIntegralBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PAdicWildFunctionFieldIntegralBasis
--E 1

)spool
)lisp (bye)

---

PAdicWildFunctionFieldIntegralBasis.help ---

====================================================================
PAdicWildFunctionFieldIntegralBasis examples
====================================================================

In this package K is a finite field, R is a ring of univariate polynomials over K, and F is a monogenic algebra over R.

We require that F is monogenic, i.e. that F = K[x,y]/(f(x,y)), because the integral basis algorithm used will factor the polynomial f(x,y). The package provides a function to compute the integral closure of R in the quotient field of F as well as a function to compute a "local integral basis" at a specific prime.

See Also:
o )show PAdicWildFunctionFieldIntegralBasis

---
PAdicWildFunctionFieldIntegralBasis (PWFFINTB)

Exports:
reducedDiscriminant integralBasis localIntegralBasis

— package PWFFINTB PAdicWildFunctionFieldIntegralBasis —

)abbrev package PWFFINTB PAdicWildFunctionFieldIntegralBasis
++ Author: Clifton Williamson
++ Date Created: 5 July 1993
++ Date Last Updated: 17 August 1993
++ Description:
++ In this package K is a finite field, R is a ring of univariate
++ polynomials over K, and F is a monogenic algebra over R.
++ We require that F is monogenic, i.e. that \( F = K[x,y]/(f(x,y)) \),
++ because the integral basis algorithm used will factor the polynomial
++ \( f(x,y) \). The package provides a function to compute the integral
closure of R in the quotient field of F as well as a function to compute
a "local integral basis" at a specific prime.

PAdicWildFunctionFieldIntegralBasis(K,R,UP,F): Exports == Implementation where
K : FiniteFieldCategory
R : UnivariatePolynomialCategory K
UP : UnivariatePolynomialCategory R
F : MonogenicAlgebra(R,UP)
I ==> Integer
L ==> List
L2 ==> ListFunctions2
Mat ==> Matrix R
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
Q ==> Fraction R
SAE ==> SimpleAlgebraicExtension
SUP ==> SparseUnivariatePolynomial
CDEN ==> CommonDenominator
DDFACT ==> DistinctDegreeFactorize
WFFINTBS => WildFunctionFieldIntegralBasis
Result  => Record(basis: Mat, basisDen: R, basisInv:Mat)
IResult  => Record(basis: Mat, basisDen: R, basisInv:Mat,discr: R)
IBPTOOLS => IntegralBasisPolynomialTools
IBACHIN  => ChineseRemainderToolsForIntegralBases
IRREDFFX => IrredPolyOverFiniteField
GHEN    => GeneralHenselPackage

Exports => with
  integralBasis : () -> Result
    ++ \spad{integralBasis()} returns a record
    ++ \spad{\{basis,basisDen,basisInv\} } containing information regarding
    ++ the integral closure of R in the quotient field of the framed
    ++ algebra F. F is a framed algebra with R-module basis
    ++ \spad{\{w1,w2,\ldots,wn\}}.
    ++ If 'basis' is the matrix \spad{\{aij, i = 1..n, j = 1..n\}}, then
    ++ the \spad{ij}th element of the integral basis is
    ++ \spad{vi = (1/basisDen) * \sum \{aij * wj, j = 1..n\}}, i.e. the
    ++ \spad{ij}th row of 'basis' contains the coordinates of the
    ++ \spad{ij}th basis vector. Similarly, the \spad{ji}th row of the
    ++ matrix 'basisInv' contains the coordinates of \spad{\{wi\}} with respect
    ++ to the basis \spad{\{v1,\ldots,vn\}}: if 'basisInv' is the matrix
    ++ \spad{\{bij, i = 1..n, j = 1..n\}}, then
    ++ \spad{wi = \sum \{bij * vj, j = 1..n\}}.

localIntegralBasis : R -> Result
  ++ \spad{integralBasis(p)} returns a record
  ++ \spad{\{basis,basisDen,basisInv\} } containing information regarding
  ++ the local integral closure of R at the prime \spad{p} in the quotient
  ++ field of the framed algebra F. F is a framed algebra with R-module
  ++ basis \spad{\{w1,w2,\ldots,wn\}}.
  ++ If 'basis' is the matrix \spad{\{aij, i = 1..n, j = 1..n\}}, then
  ++ the \spad{ij}th element of the local integral basis is
  ++ \spad{vi = (1/basisDen) * \sum \{aij * wj, j = 1..n\}}, i.e. the
  ++ \spad{ij}th row of 'basis' contains the coordinates of the
  ++ \spad{ij}th basis vector. Similarly, the \spad{ji}th row of the
  ++ matrix 'basisInv' contains the coordinates of \spad{\{wi\}} with respect
  ++ to the basis \spad{\{v1,\ldots,vn\}}: if 'basisInv' is the matrix
  ++ \spad{\{bij, i = 1..n, j = 1..n\}}, then
  ++ \spad{wi = \sum \{bij * vj, j = 1..n\}}.

reducedDiscriminant: UP -> R
  ++ reducedDiscriminant(up) \undocumented

Implementation => add
  import IntegralBasisTools(R, UP, F)
  import GeneralHenselPackage(R,UP)
  import ModularHermitianRowReduction(R)
  import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

reducedDiscriminant f ==
  ff : SUP Q := mapUnivariate((r1:R):Q+-+r1 :: Q,f)$$IBPTOOLS(R,UP,SUP UP,Q)
ee := extendedEuclidean(ff,differentiate ff)
cc := concat(coefficients(ee.coef1),coefficients(ee.coef2))
cden := splitDenominator(cc)$CDEN(R,Q,L Q)
denom := cden.den
gg := gcd map(numer,cden.num)$L2(Q,R)
(ans := denom exquo gg) case "failed" =>
  error "PWFFINTB: error in reduced discriminant computation"
ans :: R

compLocalBasis: (UP,R) -> Result
compLocalBasis(poly,prime) ==
  -- compute a local integral basis at 'prime' for k[x,y]/(poly(x,y)).
sae := SAE(R,UP,poly)
localIntegralBasis(prime)$WFFINTBS(K,R,UP,sae)

compLocalBasisOverExt: (UP,R,UP,NNI) -> Result
compLocalBasisOverExt(poly0,prime0,irrPoly0,k) ==
  -- poly0 = irrPoly0**k (mod prime0)
n := degree poly0; disc0 := discriminant poly0
(disc0 exquo prime0) case "failed" =>
  [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
r := degree irrPoly0
  -- extend scalars:
  -- construct irreducible polynomial of degree r over K
  irrPoly := generateIrredPoly(r :: PI)$IRREDFFX(K)
  -- construct extension of degree r over K
  E := SAE(K,SUP K,irrPoly)
  -- lift coefficients to elements of E
  poly := mapBivariate((k1:K):E +-> k1::E,poly0)$IBPTOOLS(K,R,UP,E)
  redDisc0 := reducedDiscriminant poly0
  redDisc := mapUnivariate((k1:K):E +-> k1::E,redDisc0)$IBPTOOLS(K,R,UP,E)
  prime := mapUnivariate((k1:K):E +-> k1::E,prime0)$IBPTOOLS(K,R,UP,E)
sae := SAE(E,SUP E,prime)
  -- reduction (mod prime) of polynomial of which poly is the kth power
  redIrrPoly :=
    pp := mapBivariate((k1:K):E +-> k1::E,irrPoly0)$IBPTOOLS(K,R,UP,E)
    mapUnivariate(reduce,pp)$IBPTOOLS(SUP E,SUP SUP E,SUP SUP SUP E,sae)
  -- factor the reduction
  factorListSAE := factors factor(redIrrPoly)$DDFACT(sae,SUP sae)
  -- list the 'primary factors' of the reduction of poly
  redFactors : List SUP sae := [(f.factor)**k for f in factorListSAE]
  -- lift these factors to elements of SUP E
  primaries : List SUP E :=
    [mapUnivariate(lift,ff)$IBPTOOLS(SUP E,SUP SUP E,SUP SUP SUP E,sae) _
     for ff in redFactors]
  -- lift the factors to factors modulo a suitable power of 'prime'
deg := (1 + order(redDisc,prime) * degree(prime)) :: PI
  henselInfo := HenselLift(poly,primaries,prime,deg)$GHEN(SUP E,SUP SUP E)
  henselFactors := henselInfo.plist
  psi1 := first henselFactors
CHAPTER 17.  CHAPTER P

FF := SAE(SUP E, SUP E, psi1)
factorIb := localIntegralBasis(prime)$WFFINTBS(E, SUP E, SUP SUP E, FF)
bs := listConjugateBases(factorIb, size()$K, r)$IBACHIN(E, SUP E, SUP SUP E)
ib := chineseRemainder(henselFactors, bs, n)$IBACHIN(E, SUP E, SUP SUP E)
b : Matrix R :=
  
b := mapMatrixIfCan(retractIfCan, ib.basis)$IBPTOOLS(K, R, UP, E)
  
  bas case "failed" => error "retraction of basis failed"
bas :: Matrix R
bInv : Matrix R :=
  
  --bas := mapMatrixIfCan(ric, ib.basisInv)$IBPTOOLS(K, R, UP, E)
  
  bas := mapMatrixIfCan(retractIfCan, ib.basisInv)$IBPTOOLS(K, R, UP, E)
  
  bas case "failed" => error "retraction of basis inverse failed"
bas :: Matrix R
bDen : R :=
  
p := mapUnivariateIfCan(retractIfCan, ib.basisDen)$IBPTOOLS(K, R, UP, E)
p case "failed" => error "retraction of basis denominator failed"
p :: R
[b, bDen, bInv]

padicLocalIntegralBasis: (UP, R, R, R) -> IResult

padicLocalIntegralBasis(p, disc, redDisc, prime) ==
  
  -- polynomials in x modulo 'prime'
  sae := SAE(K, R, prime)

  -- find the factorization of 'p' modulo 'prime' and lift the
  -- prime powers to elements of UP:
  -- reduce 'p' modulo 'prime'
  reducedP := mapUnivariate(reduce, p)$IBPTOOLS(R, UP, SUP UP, sae)

  -- factor the reduced polynomial
  factorListSAE := factors factor(reducedP)$DDFACT(sae, SUP sae)

  -- if only one prime factor, perform usual integral basis computation
  (# factorListSAE) = 1 =>
    ib := localIntegralBasis(prime)$WFFINTBS(K, R, UP, F)
    index := diagonalProduct(ib.basisInv)
    [ib.basis, ib.basisDen, ib.basisInv, disc quo (index * index)]

    -- list the 'prime factors' of the reduced polynomial
    redPrimes : List SUP sae :=
      [f.factor for f in factorListSAE]

    -- lift these factors to elements of UP
    primes : List UP :=
      [mapUnivariate(lift, ff)$IBPTOOLS(R, UP, SUP UP, sae) for ff in redPrimes]

    -- list the exponents
    expons : List NNI := [(f.exponent :: NNI) for f in factorListSAE]

    -- list the 'primary factors' of the reduced polynomial
    redPrimaries : List SUP sae :=
      [(f.factor) **((f.exponent) :: NNI) for f in factorListSAE]

    -- lift these factors to elements of UP
    primaries : List UP :=
      [mapUnivariate(lift, ff)$IBPTOOLS(R, UP, SUP UP, sae) for ff in redPrimaries]

    -- lift the factors to factors modulo a suitable power of 'prime'
    deg := (1 + order(redDisc, prime) * degree(prime)) :: PI
henselInfo := HenselLift(p, primaries, prime, deg)
henselFactors := henselInfo.plist
-- compute integral bases for the factors
factorBases : List Result := empty(); degPrime := degree prime
for pp in primes for k in expons for qq in henselFactors repeat
  base :=
    degPp := degree pp
    degPp > 1 and gcd(degPp, degPrime) = 1 =>
      compLocalBasisOverExt(qq, prime, pp, k)
      compLocalBasis(qq, prime)
  factorBases := concat(base, factorBases)
factorBases := reverse_! factorBases
ib := chineseRemainder(henselFactors, factorBases, rank()$F)$IBACHIN(K, R, UP)
index := diagonalProduct(ib.basisInv)
[ib.basis, ib.basisDen, ib.basisInv, disc quo (index * index)]

localIntegralBasis prime ==
  p := definingPolynomial()$F; disc := discriminant p
  -- disc := determinant traceMatrix()$F
  redDisc := reducedDiscriminant p
  ib := padicLocalIntegralBasis(p, disc, redDisc, prime)
  [ib.basis, ib.basisDen, ib.basisInv]

listSquaredFactors: R -> List R
listSquaredFactors px ==
  -- returns a list of the factors of px which occur with
  -- exponent > 1
  ans : List R := empty()
  factored := factor(px)$DistinctDegreeFactorize(K, R)
  for f in factors(factored) repeat
    if f.exponent > 1 then ans := concat(f.factor, ans)
  ans

integralBasis() ==
  p := definingPolynomial()$F; disc := discriminant p; n := rank()$F
  -- traceMat := traceMatrix()$F; n := rank()$F
  -- disc := determinant traceMat -- discriminant of current order
  singList := listSquaredFactors disc -- singularities of relative Spec
  redDisc := reducedDiscriminant p
  runningRb := runningRbinv := scalarMatrix(n, 1)$Mat
  -- runningRb = basis matrix of current order
  -- runningRbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  runningRbden : R := 1
  -- runningRbden = denominator for current basis matrix
  empty? singList => [runningRb, runningRbden, runningRbinv]
  for prime in singList repeat
    lb := padicLocalIntegralBasis(p, disc, redDisc, prime)
    rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
    disc := lb.dscr
\[ \text{mat} := \text{vertConcat}(\text{rbden} \times \text{runningRb}, \text{runningRbden} \times \text{rb}) \]
\[ \text{runningRbden} := \text{runningRbden} \times \text{rbden} \]
\[ \text{runningRb} := \text{squareTop rowEchelon(mat, runningRbden)} \]
\[ \text{runningRb} := \text{squareTop rowEch mat} \]
\[ \text{runningRbinv} := \text{UpTriBddDenomInv(runningRb, runningRbden)} \]
\[ [\text{runningRb, runningRbden, runningRbinv}] \]

---

--- PWFFINTB.dotabb ---

"PWFFINTB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PWFFINTB"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"PWFFINTB" -> "MONOGEN"

---

package YSTREAM ParadoxicalCombinatorsForStreams

--- ParadoxicalCombinatorsForStreams.input ---

)set break resume
)sys rm -f ParadoxicalCombinatorsForStreams.output
)spool ParadoxicalCombinatorsForStreams.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParadoxicalCombinatorsForStreams
--E 1

)spool
)lisp (bye)

---

--- ParadoxicalCombinatorsForStreams.help ---

====================================================================
ParadoxicalCombinatorsForStreams examples
====================================================================
Computation of fixed points of mappings on streams

See Also:
o  )show ParadoxicalCombinatorsForStreams

---

ParadoxicalCombinatorsForStreams (YSTREAM)

Exports:
Y

--- package YSTREAM ParadoxicalCombinatorsForStreams ---

)abbrev package YSTREAM ParadoxicalCombinatorsForStreams
++ Author: Burge, Watt (revised by Williamson)
++ Date Created: 1986
++ Date Last Updated: 21 October 1989
++ Description:
++ Computation of fixed points of mappings on streams

ParadoxicalCombinatorsForStreams(A):Exports == Implementation where
  ++ This package implements fixed-point computations on streams.
  A : Type
  ST ==> Stream
  L ==> List
  I ==> Integer

Exports ==> with
  Y: (ST A -> ST A) -> ST A
  ++ Y(f) computes a fixed point of the function f.
  Y: (L ST A -> L ST A,L) -> L ST A
  ++ Y(g,n) computes a fixed point of the function g, where g takes
++ a list of n streams and returns a list of n streams.

Implementation ==> add

\[ Y(f) = \]
\[
y : ST A := CONS(0$I,0$I)$Lisp
j := f y
RPLACA(y,frst j)$Lisp
RPLACD(y,rst j)$Lisp
y
\]

\[ Y(g,n) = \]
\[
x : L ST A := [CONS(0$I,0$I)$Lisp for i in 1..n]
j := g x
for xi in x for ji in j repeat
RPLACA(xi,frst ji)$Lisp
RPLACD(xi,rst ji)$Lisp
x
\]

package PLEQN ParametricLinearEquations

--- ParametricLinearEquations.input ---

)set break resume
)sys rm -f ParametricLinearEquations.output
)spool ParametricLinearEquations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricLinearEquations
--E 1
This package completely solves a parametric linear system of equations by decomposing the set of all parametric values for which the linear system is consistent into a union of quasi-algebraic sets (which need not be irredundant, but most of the time is). Each quasi-algebraic set is described by a list of polynomials that vanish on the set, and a list of polynomials that vanish at no point of the set. For each quasi-algebraic set, the solution of the linear system is given, as a particular solution and a basis of the homogeneous system.

The parametric linear system should be given in matrix form, with a coefficient matrix and a right hand side vector. The entries of the coefficient matrix and right hand side vector should be polynomials in the parametric variables, over a Euclidean domain of characteristic zero.

If the system is homogeneous, the right hand side need not be given. The right hand side can also be replaced by an indeterminate vector, in which case, the conditions required for consistency will also be given.

The package has other facilities for saving results to external files, as well as solving the system for a specified minimum rank. Altogether there are 12 mode maps for psolve, as explained below.

See Also:
- )show ParametricLinearEquations
ParametricLinearEquations (PLEQN)

Exports:
bsolve  B1solve  dmp2rfi  dmp2rfi  dmp2rfi
factorset hasoln  inconsistent?  maxrank  minrank
minset  nextSublist  overset?  ParCond  ParCondList
pr2dmp  psolve  rdregime  redmat  redpps
regime  se2rfi  sqfree  wrregime

-- modified to conform with new runtime system 06/04/90
-- updated with comments for MB, 02/16/94
-- also cleaned up some unnecessary arguments in regime routine
--
-- MB: In order to allow the rhs to be indeterminate, while working
-- mainly with the parametric variables on the lhs (less number of
-- variables), certain conversions of internal representation from
-- GR to Polynomial R and Fraction Polynomial R are done. At the time
-- of implementation, I thought that this may be more efficient. I
-- have not done any comparison since that requires rewriting the
-- package. My own application needs to call this package quite often,
-- and most computations involves only polynomials in the parametric
-- variables.

-- The 12 modes of psolve are probably not all necessary. Again, I
-- was thinking that if there are many regimes and many ranks, then
-- the output is quite big, and it may be nice to be able to save it
-- and read the results in later to continue computing rather than
-- recomputing. Because of the combinatorial nature of the algorithm
-- (computing all subdeterminants!), it does not take a very big matrix
-- to get into many regimes. But I now have second thoughts of this
-- design, since most of the time, the results are just intermediate,
-- passed to be further processed. On the other hand, there is probably
-- no penalty in leaving the options as is.

— package PLEQN ParametricLinearEquations —
(abbrev package PLEQN ParametricLinearEquations
++ Author: William Sit, spring 89
++ Description:
++ This package completely solves a parametric linear system of equations
++ by decomposing the set of all parametric values for which the linear
++ system is consistent into a union of quasi-algebraic sets (which need
++ not be irredundant, but most of the time is). Each quasi-algebraic
++ set is described by a list of polynomials that vanish on the set, and
++ a list of polynomials that vanish at no point of the set.
++ For each quasi-algebraic set, the solution of the linear system
++ is given, as a particular solution and a basis of the homogeneous
++ system.
++
++ The parametric linear system should be given in matrix form, with
++ a coefficient matrix and a right hand side vector. The entries
++ of the coefficient matrix and right hand side vector should be
++ polynomials in the parametric variables, over a Euclidean domain
++ of characteristic zero.
++
++ If the system is homogeneous, the right hand side need not be given.
++ The right hand side can also be replaced by an indeterminate vector,
++ in which case, the conditions required for consistency will also be
++ given.
++
++ The package has other facilities for saving results to external
++ files, as well as solving the system for a specified minimum rank.
++ Altogether there are 12 mode maps for psolve, as explained below.

ParametricLinearEquations(R,Var,Expon,GR):
  Declaration == Definition where

  R : Join(EuclideanDomain, CharacteristicZero)
  -- Warning: does not work if R is a field! because of Fraction R
  Var : Join(OrderedSet,ConvertibleTo(Symbol))
  Expon : OrderedAbelianMonoidSup
  GR : PolynomialCategory(R,Expon,Var)
  F == Fraction R
  FILE ==> FileCategory
  FNAME ==> FileName
  GB ==> EuclideanGroebnerBasisPackage
  -- GBINTERN ==> GroebnerInternalPackage
  I ==> Integer
  L ==> List
  M ==> Matrix
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  P ==> Polynomial
  PI ==> PositiveInteger
  SEG ==> Segment
  SM ==> SquareMatrix
S ==> String  
V ==> Vector  
mf ==> MultivariateFactorize(Var,Expon,R,GR)  
rp ==> GB(R,Expon,Var,GR)  
rb ==> GB(R,Expon,Var,GR)  
PR ==> PR  
GF ==> Fraction PR  
plift ==> PolynomialCategoryLifting(Expon,Var,R,GR,GF)  
Inputmode ==> Integer  
groebner ==> euclideanGroebner  
redPol ==> euclideanNormalForm

-- MB: The following macros are data structures to store mostly  
-- intermediate results  
-- Rec stores a subdeterminant with corresponding row and column indices  
-- Fgb is a Groebner basis for the ideal generated by the subdeterminants  
-- of a given rank.  
-- Linsys specifies a linearly independent system of a given system  
-- assuming a given rank, using given row and column indices  
-- Linsln stores the solution to the parametric linear system as a basis  
-- and a particular solution (for a given regime)  
-- Rec2 stores the rank, and a list of subdeterminants of that rank,  
-- and a Groebner basis for the ideal they generate.  
-- Rec3 stores a regime and the corresponding solution; the regime is  
-- given by a list of equations (eqzro) and one inequation (neqzro)  
-- describing the quasi-algebraic set which is the regime; the  
-- additional consistency conditions due to the rhs is given by wcond.  
-- Ranksolns stores a list of regimes and their solutions, and the number  
-- of regimes all together.  
-- Rec8 (temporary) stores a quasi-algebraic set with an indication  
-- whether it is empty (sysok = false) or not (sysok = true).

-- I think psolve should be renamed parametricSolve, or even  
-- parametricLinearSolve. On the other hand, may be just solve will do.  
-- Please feel free to change it to conform with system conventions.  
-- Most psolve routines return a list of regimes and solutions,  
-- except those that output to file when the number of regimes is  
-- returned instead.  
-- This version has been tested on the pc version 1.608 March 13, 1992

Rec ==> Record(det:GR,rows:L I,cols:L I)  
Eqns ==> L Rec  
Fgb ==> L GR -- groebner basis  
Linsoln ==> Record(partsol:V GF,basis:L V GF)  
Linsys ==> Record(mat:M GF,vec:L GF,rank:NNI,rows:L I,cols:L I)  
Rec2 ==> Record(rank:NNI,eqns:Eqns,fgb:Fgb)  
RankConds ==> L Rec2  
Rec3 ==> Record(eqzro:L GR, neqzro:L GR,wcond:L PR, bsoln:Linsoln)  
Ranksolns ==> Record(rgl:L Rec3,rgsz:I)  
Rec8 ==> Record(sysok:Boolean, z0:L GR, n0:L GR)
Declaration == with
  psolve: (M GR, L GR) -> L Rec3
  ++ psolve(c,w) solves c z = w for all possible ranks
  ++ of the matrix c and given right hand side vector w
  -- this is mode 1
  psolve: (M GR, L Symbol) -> L Rec3
  ++ psolve(c,w) solves c z = w for all possible ranks
  ++ of the matrix c and indeterminate right hand side w
  -- this is mode 2
  psolve: M GR -> L Rec3
  ++ psolve(c) solves the homogeneous linear system
  ++ c z = 0 for all possible ranks of the matrix c
  -- this is mode 3
  psolve: (M GR, L GR, PI) -> L Rec3
  ++ psolve(c,w,k) solves c z = w for all possible ranks >= k
  ++ of the matrix c and given right hand side vector w
  -- this is mode 4
  psolve: (M GR, L Symbol, PI) -> L Rec3
  ++ psolve(c,w,k) solves c z = w for all possible ranks >= k
  ++ of the matrix c and indeterminate right hand side w
  -- this is mode 5
  psolve: (M GR, S) -> I
  ++ psolve(c,w) solves c z = w for all possible ranks
  ++ of the matrix c and given right hand side vector w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 6
  psolve: (M GR, L Symbol, S) -> I
  ++ psolve(c,w,s) solves c z = w for all possible ranks
  ++ of the matrix c and indeterminate right hand side w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 7
  psolve: (M GR, S) -> I
  ++ psolve(c,s) solves c z = 0 for all possible ranks
  ++ of the matrix c and given right hand side vector w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 8
  psolve: (M GR, L GR, PI, S) -> I
  ++ psolve(c,w,k,s) solves c z = w for all possible ranks >= k
  ++ of the matrix c and given right hand side w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 9
CHAPTER 17. \textit{CHAPTER P}

-- this is mode 10

\texttt{psolve: (M GR, L Symbol, PI, S) \rightarrow I}
\begin{itemize}
  \item \texttt{psolve(c,w,k,s)} solves \( c z = w \) for all possible ranks \( \geq k \)
  \item \texttt{writes the results to a file named s, and returns the}
  \item \texttt{number of regimes}
\end{itemize}

-- this is mode 11

\texttt{psolve: (M GR, PI, S) \rightarrow I}
\begin{itemize}
  \item \texttt{psolve(c,k,s)} solves \( c z = 0 \) for all possible ranks \( \geq k \)
  \item \texttt{writes the results to a file named s, and returns the}
  \item \texttt{number of regimes}
\end{itemize}

-- this is mode 12

\texttt{wrregime : (L Rec3, S) \rightarrow I}
\begin{itemize}
  \item \texttt{wrregime(l,s)} writes a list of regimes to a file named s
  \item \texttt{and returns the number of regimes written}
\end{itemize}

\texttt{rdregime : S \rightarrow L Rec3}
\begin{itemize}
  \item \texttt{rdregime(s)} reads in a list from a file with name s
\end{itemize}

-- for internal use only --

-- these are exported so my other packages can use them

\texttt{bsolve: (M GR, L GF, NNI, S, Inputmode) \rightarrow Ranksolns}
\begin{itemize}
  \item \texttt{bsolve(c, w, r, s, m)} returns a list of regimes and
  \item \texttt{solutions of the system c z = w for ranks at least r;}
  \item \texttt{depending on the mode m chosen, it writes the output to}
  \item \texttt{a file given by the string s.}
\end{itemize}

\texttt{dmp2rfi: GR \rightarrow GF}
\begin{itemize}
  \item \texttt{dmp2rfi(p)} converts \( p \) to target domain
\end{itemize}

\texttt{dmp2rfi: M GR \rightarrow M GF}
\begin{itemize}
  \item \texttt{dmp2rfi(m)} converts \( m \) to target domain
\end{itemize}

\texttt{dmp2rfi: L GR \rightarrow L GF}
\begin{itemize}
  \item \texttt{dmp2rfi(l)} converts \( l \) to target domain
\end{itemize}

\texttt{se2rfi: L Symbol \rightarrow L GF}
\begin{itemize}
  \item \texttt{se2rfi(l)} converts \( l \) to target domain
\end{itemize}

\texttt{pr2dmp: PR \rightarrow GR}
\begin{itemize}
  \item \texttt{pr2dmp(p)} converts \( p \) to target domain
\end{itemize}

\texttt{hasoln: (Fgb, L GR) \rightarrow Rec8}
\begin{itemize}
  \item \texttt{hasoln(g, l)} tests whether the quasi-algebraic set
  \item \texttt{defined by p = 0 for p in g and q ^= 0 for q in l}
  \item \texttt{is empty or not and returns a simplified definition}
  \item \texttt{of the quasi-algebraic set}
  \item -- this is now done in QALGSET package
\end{itemize}

\texttt{ParCondList: (M GR, NNI) \rightarrow RankConds}
\begin{itemize}
  \item \texttt{ParCondList(c,r)} computes a list of subdeterminants of each
  \item \texttt{rank \( \geq r \) of the matrix c and returns}
  \item \texttt{a groebner basis for the}
  \item \texttt{ideal they generate}
\end{itemize}

\texttt{redpps: (Linsoln, Fgb) \rightarrow Linsoln}
++ redpps(s,g) returns the simplified form of s after reducing
++ modulo a groebner basis g

--- LOCAL FUNCTIONS ---

B1solve: Linsys -> Linsoln
++ B1solve(s) solves the system (s.mat) z = s.vec
++ for the variables given by the column indices of s.cols
++ in terms of the other variables and the right hand side s.vec
++ by assuming that the rank is s.rank,
++ that the system is consistent, with the linearly
++ independent equations indexed by the given row indices s.rows;
++ the coefficients in s.mat involving parameters are treated as
++ polynomials. B1solve(s) returns a particular solution to the
++ system and a basis of the homogeneous system (s.mat) z = 0.

factorset: GR -> L GR
++ factorset(p) returns the set of irreducible factors of p.

maxrank: RankConds -> NNI
++ maxrank(r) returns the maximum rank in the list r of regimes

minrank: RankConds -> NNI
++ minrank(r) returns the minimum rank in the list r of regimes

minset: L L GR -> L L GR
++ minset(sl) returns the sublist of sl consisting of the minimal
++ lists (with respect to inclusion) in the list sl of lists

nextSublist: (I, I) -> L L I
++ nextSublist(n,k) returns a list of k-subsets of {1, ..., n}.

overset?: (L GR, L L GR) -> Boolean
++ overset?(s,sl) returns true if s properly a sublist of a member
++ of sl; otherwise it returns false

ParCond : (M GR,NNI) -> Eqns
++ ParCond(m,k) returns the list of all k by k subdeterminants in
++ the matrix m

redmat: (M GR, Fgb) -> M GR
++ redmat(m,g) returns a matrix whose entries are those of m
++ modulo the ideal generated by the groebner basis g

regime: (Rec,M GR,L GF,L L GR,NNI,NNI,Inputmode) -> Rec3
++ regime(y,c, w, p, r, rm, m) returns a regime,
++ a list of polynomials specifying the consistency conditions,
++ a particular solution and basis representing the general
++ solution of the parametric linear system c z = w
++ on that regime. The regime returned depends on
++ the subdeterminant y.det and the row and column indices.
++ The solutions are simplified using the assumption that
++ the system has rank r and maximum rank rm. The list p
++ represents a list of list of factors of polynomials in
++ a groebner basis of the ideal generated by higher order
++ subdeterminants, and ius used for the simplification.
++ The mode m
++ distinguishes the cases when the system is homogeneous,
++ or the right hand side is arbitrary, or when there is no
++ new right hand side variables.
sqfree: GR -> GR
++ sqfree(p) returns the product of square free factors of p
inconsistent?: L GR -> Boolean
++ inconsistent?(pl) returns true if the system of equations
++ p = 0 for p in pl is inconsistent. It is assumed
++ that pl is a groebner basis.
-- this is needed because of change to
-- EuclideanGroebnerBasisPackage
inconsistent?: L PR -> Boolean
++ inconsistent?(pl) returns true if the system of equations
++ p = 0 for p in pl is inconsistent. It is assumed
++ that pl is a groebner basis.
-- this is needed because of change to
-- EuclideanGroebnerBasisPackage

Definition == add

inconsistent?(pl:L GR):Boolean ==
  for p in pl repeat
    ground? p => return true
  false
inconsistent?(pl:L PR):Boolean ==
  for p in pl repeat
    ground? p => return true
  false

B1solve (sys:Linsys):Linsoln ==
i,j,i1,j1:I
rss:L I:=sys.rows
nss:L I:=sys.cols
k:=sys.rank
cmat:M GF:=sys.mat
n:=ncols cmat
frcols:L I:=setDifference$(L I) (expand$(SEG I) (1..n), nss)
w:L GF:=sys.vec
p:V GF:=new(n,0)
pbas:L V GF:=[]
if k "= 0 then
  augmat:M GF:=zero(k,n+1)
  for i in rss for i1 in 1.. repeat
    for j in nss for j1 in 1.. repeat
      augmat(i1,j1):=cmat(i,j)
  for j in frcols for j1 in k+1.. repeat
    augmat(i1,j1):=-cmat(i,j)
  augmat(i1,n+1):=w.i
  augmat:=rowEchelon$(M GF) augmat
  for i in nss for ii in 1.. repeat p.i:=augmat(ii,n+1)
for j in frcols for j1 in k+1.. repeat
  pb:V GF:=new(n,0)
  pb.j:=1
  for i in nss for i1 in 1.. repeat
    pb.i:=augmat(i1,j1)
  pbas:=cons(pb,pbas)
else
  for j in frcols for j1 in k+1.. repeat
    pb:V GF:=new(n,0)
    pb.j:=1
    pbas:=cons(pb,pbas)
  [p,pbas]

regime (y, coef, w, psbf, rk, rkmax, mode) ==
  i,j:I
  -- use the y.det nonzero to simplify the groebner basis
  -- of ideal generated by higher order subdeterminants
  ydetf:L GR:=factorset y.det
  yzero:L GR:=
  rk = rkmax => nil$GR
  psbf:=[setDifference(x, ydetf) for x in psbf]
  groebner$gb */x for x in psbf
  -- simplify coefficients by modulo ideal
  nc:M GF:=dmp2rfi redmat(coef,yzero)
  -- solve the system
  rss:L I:=y.rows; nss:L I :=y.cols
  sys:Linsys:=[nc,w,rk,rss,nss]\$Linsys
  pps: B1solve(sys)
  pp:=pps.partsol
  frows:L I:=setDifference$(L I) (expand$(SEG I) (1..nrows coef),rss)
  wcd:L PR:=
  -- case homogeneous rhs
  entry? (mode, [3,6,9,12]$(L I)) =>
    [yzero, ydetf, wcd, redpps(pps, yzero)]$Rec3
  -- case arbitrary rhs, pps not reduced
  for i in frows repeat
    weqn:GF:=[nc(i,j)*pp.j for j in nss]
    wnum:PR:=numer$GF (w.i - weqn)
    wnum = 0 => "trivially satisfied"
    ground? wnum => return [yzero, ydetf,[1]\$PR,pps]$Rec3
    wcd:=cons(wnum,wcd)
  entry? (mode, [2,5,8,11]$(L I)) => [yzero, ydetf, wcd, pps]$Rec3
  -- case no new rhs variable
  if not empty? wcd then _
    yzero:=removeDuplicates append(yzero,[pr2dmp pw for pw in wcd])
  test:Rec8:=hasoln (yzero, ydetf)
  not test.sysok => [test.z0, test.n0, [1]\$PR,pps]$Rec3
  [test.z0, test.n0, [], redpps(pps, test.z0)]$Rec3

bsolve (coeff, w, h, outname, mode) ==
r:=nrows coeff
-- n:=ncols coeff
r ^= #w => error "number of rows unequal on lhs and rhs"
newfile:FNAME
rkso1n:File Rec3
count:I:=0
lrec3:L Rec3:=[]
filename:Boolean:= entry? (mode, [7,8,9,10,11,12]$(L I))
if filename then
    newfile:=new$FNAME ("", outname,"regime")
rkso1n:=open$(File Rec3) newfile
y:Rec
k:NNI
rkcl:RankConds:=
    entry? (mode,[1,2,3,7,8,9]$(L I)) => ParCondList (coeff,0)
    entry? (mode,[4,5,6,10,11,12]$(L I)) => ParCondList (coeff,h)
rkmax:=maxrank rrcl
rkmin:=minrank rrcl
for k in rkmax-rkmin+1..1 by -1 repeat
    rk:=rrcl.k.rank
    pc:Eqns:=rrcl.k.eqns
    psb:Fgb:=(if rk=rkmax then [] else rrcl.(k+1).fgb)
    psbf:L L GR:= [factorset x for x in psb]
    psbf:= minset(psbf)
    for y in pc repeat
        rec3:Rec3:= regime (y, coeff, w, psbf, rk, rkmax, mode)
        inconsistent? rec3.wcond => "incompatible system"
        if filename then write_! (rkso1n, rec3)
        else lrec3:= cons(rec3, lrec3)
        count:=count+1
        if filename then close_! rkso1n
    [lrec3, count]$Ranksolns
factorset y ==
ground? y => []
[j.factor for j in factors(factor$mf y)]
ParCondList (mat, h) ==
rcl: RankConds:= []
ps: L GR:=[]
pc:Eqns:=[]
npc: Eqns:=[]
psbf: Fgb:=[]
rc: Rec
done: Boolean := false
r:=nrows mat
n:=ncols mat
maxrk:I:=min(r,n)
k:NNI
for k in min(r,n)..h by -1 until done repeat
pc := ParCond(mat, k)
npc := []

(empty? pc) and (k >= 1) => maxrk := k - 1

if ground? pc.1.det -- only one is sufficient (neqzro = {})
then (npc := pc; done := true; ps := [1$GR])
else

zro := (if k = maxrk then [] else rcl.1.fgb)
covered := false

for rc in pc until covered repeat
p := redPol$rp (rc.det, zro)
p = 0 => "incompatible or covered subdeterminant"
test := hasoln(zro, [rc.det])

-- zroideal := ideal(zro)
-- inRadical? (p, zroideal) => "incompatible or covered"
"test.sysok => "incompatible or covered"

-- The next line is WRONG! cannot replace zro by test.z0
-- zro := groebner$gb (cons(*/test.n0, test.z0))
zro := groebner$gb (cons(p, zro))

npc := cons(rc, npc)
done := covered := inconsistent? zro

ps := zro

pcl := Rec2 := construct(k, npc, ps)
rcl := cons(pcl, rcl)

rcl

redpps(pps, zz) ==

pv := pps.partsol
r := #pv
pb := pps.basis
n := #pb + 1

nummat := zero(r, n)
denmat := zero(r, n)

for i in 1..r repeat

nummat(i, 1) := pr2dmp numer$GF pv.i
denmat(i, 1) := pr2dmp denom$GF pv.i

for j in 2..n repeat

nummat(i, j) := pr2dmp numer$GF (pb.(j-1)).i
denmat(i, j) := pr2dmp denom$GF (pb.(j-1)).i

nummat := redmat(nummat, zz)
denmat := redmat(denmat, zz)

for i in 1..r repeat

pv.i := (dmp2rfi nummat(i, 1))/(dmp2rfi denmat(i, 1))

for j in 2..n repeat

pbj := new(r, 0)

for i in 1..r repeat

pbj.i := (dmp2rfi nummat(i, j))/(dmp2rfi denmat(i, j))
pb.(j-1) := pbj
dmp2rfi (mat:M GF): M GF ==
r:=nrows mat
n:=ncols mat
nmat:=zero(r,n)
for i in 1..r repeat
  for j in 1..n repeat
    nmat(i,j):=dmp2rfi mat(i,j)
nmat

dmp2rfi (vl: L GF):L GF ==
  [dmp2rfi v for v in vl]

psolve (mat:M GR, w:L GR): L Rec3 ==
  bsolve(mat, dmp2rfi w, 1, "nofile", 1).rgl
psolve (mat:M GR, w:L Symbol): L Rec3 ==
  bsolve(mat, se2rfi w, 1, "nofile", 2).rgl
psolve (mat:M GR): L Rec3 ==
  bsolve(mat, [0$GF for i in 1..nrows mat], 1, "nofile", 3).rgl

psolve (mat:M GR, w:L GR, h:PI): L Rec3 ==
  bsolve(mat, dmp2rfi w, h::NNI, "nofile", 4).rgl
psolve (mat:M GR, w:L Symbol, h:PI): L Rec3 ==
  bsolve(mat, se2rfi w, h::NNI, "nofile", 5).rgl
psolve (mat:M GR, h:PI): L Rec3 ==
  bsolve(mat, [0$GF for i in 1..nrows mat], h::NNI, "nofile", 6).rgl

psolve (mat:M GR, w:L GR, outname:S): I ==
  bsolve(mat, dmp2rfi w, 1, outname, 7).rgsz
psolve (mat:M GR, w:L Symbol, outname:S): I ==
  bsolve(mat, se2rfi w, 1, outname, 8).rgsz
psolve (mat:M GR, outname:S): I ==
  bsolve(mat, [0$GF for i in 1..nrows mat], 1, outname, 9).rgsz

nextSublist (n,k) ==
  n <= 0 => []
k <= 0 => [ nil$(List Integer) ]
k > n => []
n = 1 and k = 1 => [[1]]
mslist: L L I:=[[]]
for ms in nextSublist(n-1,k-1) repeat
  mslist:=cons(append(ms,[n]),mslist)
append(nextSublist(n-1,k), mslist)

psolve (mat:M GR, w:L GR, h:PI, outname:S): I ==
  bsolve(mat, dmp2rfi w, h::NNI, outname, 10).rgsz
psolve (mat:M GR, w:L Symbol, h:PI, outname:S): I ==
  bsolve(mat, se2rfi w, h::NNI, outname, 11).rgsz
psolve (mat:M GR, h:PI, outname:S): I ==
  bsolve(mat,[0$GF for i in 1..nrows mat],h::NNI,outname, 12).rgsz
hasoln (zro, nzro) ==
  empty? zro => [true, zro, nzro]
  zro:=groebner$gb zro
  inconsistent? zro => [false, zro, nzro]
  empty? nzro => [true, zro, nzro]
  pnzro: GR:=redPol$rp (*/nzro, zro)
  pnzro = 0 => [false, zro, nzro]
  nzro:=factorset pnzro
  psbf: L L GR:=minset [factorset p for p in zro]
  psbf:= [setDifference(x, nzro) for x in psbf]
  entry? ([], psbf) => [false, zro, nzro]
  zro:=groebner$gb [*/x for x in psbf]
  inconsistent? zro => [false, zro, nzro]
  nzro:=[redPol$rp (p,zro) for p in nzro]
  nzro:=[p for p in nzro | ~(ground? p)]
  [true, zro, nzro]

se2rfi w == [coerce$GF monomial$PR (1$PR, wi, 1) for wi in w]

pr2dmp p ==
  ground? p => (ground p)::GR
  algCoerceInteractive(p,PR,GR)$(Lisp) pretend GR

wrregime (lrec3, outname) ==
  newfile:FNAME:=new$FNAME ("", outname,"regime")
  rksoln: File Rec3:=open$(File Rec3) newfile
  count:I:=0 -- number of distinct regimes
  -- rec3: Rec3
  for rec3 in lrec3 repeat
    write_!(rksoln, rec3)
    count:=count+1
  close_!(rksoln)
  count

dmp2rfi (p:GR):GF ==
  map$plift ((v1:Var):GF +-> (convert v1)@Symbol::GF,
             (r1:R):GF +-> r1::PR::GF, p)

rdregime inname ==
  infilename:=filename$FNAME ("", inname, "regime")
  infile: File Rec3:=open$(File Rec3) (infilename, "input")
  rksoln:L Rec3:=[]
  rec3:Union(Rec3, "failed"):=readIfCan_!$(File Rec3) (infilename)
  while rec3 case Rec3 repeat
    rksoln:=cons(rec3::Rec3,rksoln) -- replace : to :: for AIX
    rec3:=[readIfCan_!$(File Rec3) (infilename)
  close_!(infilename)
\[
\text{rkso1n}
\]

\[
\text{maxrank } rcl &= \begin{cases} 
0 & \text{empty? } rcl \\
\text{"max"}/[j.\text{rank for } j \text{ in } rcl] & \text{otherwise}
\end{cases}
\]

\[
\text{minrank } rcl &= \begin{cases} 
0 & \text{empty? } rcl \\
\text{"min"}/[j.\text{rank for } j \text{ in } rcl] & \text{otherwise}
\end{cases}
\]

\[
\text{minset } lset &= \begin{cases} 
lset & \text{empty? } lset \\
[x \text{ for } x \text{ in } lset | \sim (\text{overset?}(x,lset))] & \text{otherwise}
\end{cases}
\]

\[
\text{sqfree } p &= \star/[j.\text{factor for } j \text{ in factors(squareFree } p)]
\]

\[
\text{ParCond } (\text{mat, } k) &= \begin{cases} 
[1, [], []] & k = 0 \\
\text{error } \text{"k exceeds maximum possible rank"} & \text{otherwise}
\end{cases}
\]

\[
\text{overset?}(p, \text{qlist}) &= \begin{cases} 
\text{false} & \text{empty? } \text{qlist} \\
\text{false} \lor \bigl[\text{brace (Set GR) q} < \text{brace (Set GR) p} \bigr] & \text{otherwise}
\end{cases}
\]

\[
\text{redmat } (\text{mat, psb}) &= \begin{cases} 
i, j: \text{I} \\
r := \text{nrows}(\text{mat}) \\
n := \text{ncols}(\text{mat}) \\
\text{newmat: M GR:=zero} (r, n)
\end{cases}
\]
for i in 1..r repeat
  for j in 1..n repeat
    p:GR:=mat(i,j)
    ground? p => newmat(i,j):=p
    newmat(i,j):=redPol$rp (p,psb)

package PARPC2 ParametricPlaneCurveFunctions2

--- ParametricPlaneCurveFunctions2.input ---

)set break resume
)sys rm -f ParametricPlaneCurveFunctions2.output
)spool ParametricPlaneCurveFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricPlaneCurveFunctions2
--E 1

)spool
)lisp (bye)

---

--- ParametricPlaneCurveFunctions2.help ---

====================================================================
ParametricPlaneCurveFunctions2 examples
====================================================================
This package has no description

See Also:
  o )show ParametricPlaneCurveFunctions2

---

**ParametricPlaneCurveFunctions2 (PARPC2)**

Exports:

map

---

)`abbrev package PARPC2 ParametricPlaneCurveFunctions2
++ Description:
++ This package has no description

ParametricPlaneCurveFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricPlaneCurve(CF1)) -> ParametricPlaneCurve(CF2)
  ++ map(f,x) \undocumented
  == add
  map(f, c) == curve(f coordinate(c,1), f coordinate(c, 2))

---

--- PARPC2.dotabb ---

"PARPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARPC2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARPC2" -> "TYPE"
package PARSC2 ParametricSpaceCurveFunctions2

--- ParametricSpaceCurveFunctions2.input ---

)set break resume
(sys rm -f ParametricSpaceCurveFunctions2.output
)spool ParametricSpaceCurveFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricSpaceCurveFunctions2
--E 1

)spool
)lisp (bye)

--- ParametricSpaceCurveFunctions2.help ---

====================================================================
ParametricSpaceCurveFunctions2 examples
====================================================================

This package has no description

See Also:
  o )show ParametricSpaceCurveFunctions2
### ParametricSpaceCurveFunctions2 (PARSC2)

**Exports:**
- `map`

---

```lisp
)abbrev package PARSC2 ParametricSpaceCurveFunctions2
++ Description:
++ This package has no description

ParametricSpaceCurveFunctions2(CF1: Type, CF2:Type): with

  map: (CF1 -> CF2, ParametricSpaceCurve(CF1)) -> ParametricSpaceCurve(CF2)
    ++ map(f,x) \undocumented

  == add
  
    map(f, c) == curve(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))
```

---

```bash
"PARSC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARSC2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSC2" -> "TYPE"
```

---

### package PARSU2 ParametricSurfaceFunctions2

---

```bash
package PARSU2 ParametricSurfaceFunctions2
```

---

`ParametricSurfaceFunctions2.input`


---

--- ParametricSurfaceFunctions2.help ---

ParametricSurfaceFunctions2 examples

This package has no description

See Also:
  o )show ParametricSurfaceFunctions2

-----

ParametricSurfaceFunctions2 (PARSU2)

Exports:
  map
— package PARSU2 ParametricSurfaceFunctions2 —

)abbrev package PARSU2 ParametricSurfaceFunctions2
++ Description:
++ This package has no description

ParametricSurfaceFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricSurface(CF1)) -> ParametricSurface(CF2)
  ++ map(f,x) undocumented
== add
  map(f, c) == surface(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))

——

— PARSU2.dotabb —

"PARSU2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARSU2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSU2" -> "TYPE"

——

package PARAMP ParametrizationPackage

— ParametrizationPackage.input —

)set break resume
)sys rm -f ParametrizationPackage.output
)spool ParametrizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametrizationPackage
--R
--R ParametrizationPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: ... ProjectiveSpaceCategory(K),PCS: LocalPowerSeriesCategory(K),Plc: PlacesCategory(K,PCS)) is a package constructor
--R Abbreviation for ParametrizationPackage is PARAMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARAMP
--R
--R-------------------------- Operations --------------------------
--R parametrize : (PolyRing,List(PCS)) -> PCS
ParametrizationPackage (PARAMP)

---

ParametrizationPackage examples

The following is part of the PAFF package

See Also:
- )show ParametrizationPackage

---
ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc):Exp == Impl where
K:Field
symb: List(Symbol)

E : DirectProductCategory(#symb,NonNegativeInteger)
OV ==> OrderedVariableList(symb)

PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)

Exp ==> with

parametrize: (PolyRing,List(PCS)) -> PCS
parametrize: (PolyRing,Plc) -> PCS
++ parametrize(f,pl) returns the local parametrization of the
++ polynomial function f at the place pl. Note that local
++ parametrization of the place must have first been compute and set.
++ For simple point on a curve, this done with \spad{pointToPlace}.
++ The local parametrization places corresponding to a leaf in a
++ desingularization tree are compute at the moment of
++ their "creation". (See package \spad{DesingTreePackage}.
parametrize: (PolyRing,PolyRing,Plc) -> PCS
++ parametrize(f,g,pl) returns the local parametrization of the
++ rational function f/g at the place pl. Note that local
++ parametrization of the place must have first been compute and set.
++ For simple point on a curve, this done with \spad{pointToPlace}.
++ The local parametrization places corresponding to a leaf in a
++ desingularization tree are compute at the moment of
++ their "creation". (See package \spad{DesingTreePackage}.
parametrize: (PolyRing,Plc,Integer) -> PCS
++ parametrize(f,pl,n) returns t**n * parametrize(f,p).

Impl => add

import PCS
import PolyRing

-- the following returns the parametrization in term of
-- the precomputed local parametrization
-- of the point pt. Note if pl is a place and pl = pt::Plc then
-- parametrize(f,pt) <> parametrize(pl) unless pt is a simple point
parametrize(f:PolyRing,localPar:List(PCS)) ==
  zero?(f) => 0
  lc:K := leadingCoefficient(f)
  ld:E := degree f
  ldp:List NonNegativeInteger := parts(ld)
  if empty?(localPar) then error _
      "the parametrization of the place or leaf has not been done yet!"
  monoPar:PCS := reduce("*", [ s**e for s in localPar for e in ldp])
  lc* monoPar + parametrize(reductum(f),localPar)

parametrize(f:PolyRing,pt:Plc) ==
  zero?(f) => 0
  localPar:List PCS := localParam pt
  parametrize(f,localPar)

parametrize(f:PolyRing,g:PolyRing,pt:Plc) ==
  sf := parametrize(f,pt)
  sg := parametrize(g,pt)
  sf * inv sg

parametrize(f:PolyRing,pt:Plc,n:Integer) ==
  s := parametrize(f,pt)
  shift(s,n)

---

package PFRPAC PartialFractionPackage

--- PartialFractionPackage.input ---

)set break resume
)spool PartialFractionPackage.output
)set message test on
)set message auto off
\( a = \frac{x+1}{y+1} \)

\[
\begin{align*}
\text{Type: Fraction(Polynomial(Integer))} \\
\end{align*}
\]

\[ b = \frac{y+1}{x+1} \]

\[
\begin{align*}
\text{Type: Fraction(Polynomial(Integer))} \\
\end{align*}
\]
The package PartialFractionPackage gives an easier to use interface to the domain PartialFraction. The user gives a fraction of polynomials, and a variable and the package converts it to the proper datatype for the PartialFraction domain.

When Axiom is given an expression such as:

\[
a := x + 1 / (y + 1)
\]

it will end up with the result in Fraction Polynomial Integer.

\[
\frac{x + x + 1}{y + 1}
\]

We might want it in a partial fraction form. This can be done with:

\[
\text{partialFraction}(a, y) \text{ PartialFractionPackage}
\]

\[
\frac{1}{y + 1}
\]

We can do the same thing with a different variable:

\[
b := y + 1 / (x + 1)
\]

\[
\frac{(x + 1)y + 1}{x + 1}
\]

\[
\text{partialFraction}(b, x) \text{ PartialFractionPackage}
\]

\[
\frac{1}{x + 1}
\]

See Also:
o )show PartialFractionPackage
PartialFractionPackage (PFRPAC)

Exports:
partialFraction

--- package PFRPAC PartialFractionPackage ---

)abbrev package PFRPAC PartialFractionPackage
++ Author: Barry M. Trager
++ Date Created: 1992
++ Description:
++ The package \spadtype{PartialFractionPackage} gives an easier
++ to use interfact the domain \spadtype{PartialFraction}.
++ The user gives a fraction of polynomials, and a variable and
++ the package converts it to the proper datatype for the
++ \spadtype{PartialFraction} domain.

PartialFractionPackage(R): Cat == Capsule where
-- R : UniqueFactorizationDomain -- not yet supported
R : Join(EuclideanDomain, CharacteristicZero)
FPR ==> Fraction Polynomial R
INDE ==> IndexedExponents Symbol
PR ==> Polynomial R
SUP ==> SparseUnivariatePolynomial
Cat == with
    partialFraction: (FPR, Symbol) -> Any
    ++ partialFraction(rf, var) returns the partial fraction decomposition
    ++ of the rational function rf with respect to the variable var.
    ++
    ++X a:=x+1/(y+1)
    ++X partialFraction(a,y)$PFRPAC(INT)
    partialFraction: (PR, Factored PR, Symbol) -> Any
    ++ partialFraction(num, facdenom, var) returns the partial fraction
    ++ decomposition of the rational function whose numerator is num and
    ++ whose factored denominator is facdenom with respect to the
    ++ variable var.
Capsule == add
partialFraction(rf, v) ==
  df := factor(denom rf)$MultivariateFactorize(Symbol, INDE,R,PR)
  partialFraction(numer rf, df, v)

makeSup(p:Polynomial R, v:Symbol) : SparseUnivariatePolynomial FPR ==
  up := univariate(p,v)
  map((z1:PR):FPR -> z1::FPR,up)
  $UnivariatePolynomialCategoryFunctions2(PR, SUP PR, FPR, SUP FPR)

partialFraction(p, facq, v) ==
  up := UnivariatePolynomial(v, Fraction Polynomial R)
  fup := Factored up
  ffact : List(Record(irr:up,pow:Integer))
  ffact:=[makeSup(u.factor,v) pretend up,u.exponent]
    for u in factors facq
  fcont:=makeSup(unit facq,v) pretend up
  nflist:fup := fcont*(*/[primeFactor(ff.irr,ff.pow) for ff in ffact])
  pfup:=partialFraction(makeSup(p,v) pretend up,nflist)$PartialFraction(up)
  coerce(pfup)$AnyFunctions1(PartialFraction up)

package PARTPERM PartitionsAndPermutations

--- PartitionsAndPermutations.input ---

)set break resume
)sys rm -f PartitionsAndPermutations.output
)spool PartitionsAndPermutations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PartitionsAndPermutations
--E 1
PartitionsAndPermutations (PARTPERM)

Exports:
  conjugate conjugates partitions permutations sequences
  shuffle shufflein

— package PARTPERM PartitionsAndPermutations —

)abbrev package PARTPERM PartitionsAndPermutations
++ Author: William H. Burge
++ Date Created: 29 October 1987
++ Date Last Updated: 3 April 1991
++ Description:
++ PartitionsAndPermutations contains functions for generating streams of
++ integer partitions, and streams of sequences of integers
++ composed from a multi-set.

PartitionsAndPermutations: Exports == Implementation where
I ==> Integer
L ==> List
ST ==> Stream
ST1 ==> StreamFunctions1
ST2 ==> StreamFunctions2
ST3 ==> StreamFunctions3

Exports ==> with

partitions: (I,I,I) -> ST L I
++ \spad{partitions(p,l,n)} is the stream of partitions
++ of n whose number of parts is no greater than p
++ and whose largest part is no greater than l.
partitions: I -> ST L I
++ \spad{partitions(n)} is the stream of all partitions of n.
partitions: (I,I) -> ST L I
++ \spad{partitions(p,l)} is the stream of all
++ partitions whose number of
++ parts and largest part are no greater than p and l.
conjugate: L I -> L I
++ \spad{conjugate(pt)} is the conjugate of the partition pt.
conjugates: ST L I -> ST L I
++ \spad{conjugates(lp)} is the stream of conjugates of a stream
++ of partitions lp.
shuffle: (L I,L I) -> ST L I
++ \spad{shuffle(l1,l2)} forms the stream of all shuffles of l1
++ and l2, i.e. all sequences that can be formed from
++ merging l1 and l2.
shufflein: (L I,ST L I) -> ST L I
++ \spad{shufflein(l,u)} maps shuffle(l,u) on to all
++ members u of st, concatenating the results.
sequences: (L I,L I) -> ST L I
++ \spad{sequences(l1,l2)} is the stream of all sequences that
++ can be composed from the multiset defined from
++ two lists of integers l1 and l2.
++ For example, the pair \spad{[[1,2,4],[2,3,5]]} represents
++ multi-set with 1 \spad{1}'s, 2 \spad{2}'s, and 4 \spad{5}'s.
sequences: L I -> ST L I
++ \spad{sequences([l0,l1,l2,...,ln])} is the set of
++ all sequences formed from
++ \spad{0}'s, \spad{1}'s, \spad{2}'s,..., \spad{ln} n's.
permutations: I -> ST L I
++ \spad{permutations(n)} is the stream of permutations
++ formed from \spad{1,2,3,...,n}.
Implementation ==> add

partitions(M,N,n) ==
  zero? n => concat(empty()$L(I),empty()$(ST L I))
  zero? M or zero? N or n < 0 => empty()
  c := map((l1:List(I)):List(I)+->concat(N,l1),partitions(M - 1,N,n - N))
  concat(c,partitions(M,N - 1,n))

partitions n == partitions(n,n,n)

partitions(M,N)==
  aaa : L ST L I := [partitions(M,N,i) for i in 0..M*N]
  concat(aaa :: ST ST L I)$ST1(L I)

-- nogreq(n,l) is the number of elements of l that are greater or
-- equal to n
nogreq: (I,L I) -> I
nogreq(n,x) == +/[1 for i in x | i >= n]

conjugate x ==
  empty? x => empty()
  [nogreq(i,x) for i in 1..first x]

conjugates z == map(conjugate,z)

shuffle(x,y)==
  empty? x => concat(y,empty())$(ST L I)
  empty? y => concat(x,empty())$(ST L I)
  concat(map((l1:List(I)):List(I)+->concat(first x,l1),shuffle(rest x,y)),
  map((l2:List(I)):List(I)+->concat(first y,l2),shuffle(x,rest y)))

shufflein(x,yy) ==
  concat(map((l1:List(I)):ST(L I)+->shuffle(x,l1),yy)$ST2(L I,ST L I))$ST1(L I)

-- rpt(n,m) is the list of n m's
rpt: (I,I) -> L I
rpt(n,m) == [m for i in 1..n]

-- zrpt(x,y) where x is [x0,x1,x2...] and y is [y0,y1,y2...]
-- is the stream [rpt(x0,y0),rpt(x1,y1),...]
zrpt: (L I,L I) -> ST L I
zrpt(x,y) == map(rpt,x :: ST I,y :: ST I)$ST3(I,I,L I)

sequences(x,y) ==
  reduce(concat(empty()$L(I),empty()$(ST L I)),
  shufflein,zrpt(x,y))$ST2(L I,ST L I)

sequences x == sequences(x,[i for i in 0..#x-1])
permutations n == sequences(rpt(n,1),[i for i in 1..n])

---

- **PARTPERM.dotabb**

"PARTPERM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARTPERM"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PARTPERM" -> "FLAGG"

---

**package PATTERN1 PatternFunctions1**

- **PatternFunctions1.input**

)set break resume
)sys rm -f PatternFunctions1.output
)spool PatternFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternFunctions1
--E 1

)spool
)lisp (bye)

---

- **PatternFunctions1.help**

============================================= PatternFunctions1 examples
=============================================

Utilities for handling patterns

See Also:
  o )show PatternFunctions1
PatternFunctions1 (PATTERN1)

Exports:
badValues addBadValue predicate satisfy? suchThat

— package PATTERN1 PatternFunctions1 —

)`abbrev package PATTERN1 PatternFunctions1
++ Author:  Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 5 Jul 1990
++ Description:
++ Utilities for handling patterns

PatternFunctions1(R:SetCategory, D:Type): with
suchThat : (Pattern R, D -> Boolean) -> Pattern R
  ++ suchThat(p, f) makes a copy of p and adds the predicate
  ++ f to the copy, which is returned.
suchThat : (Pattern R, List(D -> Boolean)) -> Pattern R
  ++ \spad{suchThat(p, [f1,...,fn])} makes a copy of p and adds the
  ++ predicate f1 and ... and fn to the copy, which is returned.
suchThat : (Pattern R, List Symbol, List D -> Boolean) -> Pattern R
  ++ \spad{suchThat(p, [a1,...,an], f)} returns a copy of p with
  ++ the top-level predicate set to \spad{f(a1,...,an)}.
predicate : Pattern R -> (D -> Boolean)
  ++ predicate(p) returns the predicate attached to p, the
  ++ constant function true if p has no predicates attached to it.
satisfy? : (D, Pattern R) -> Boolean
  ++ satisfy?(v, p) returns f(v) where f is the predicate
  ++ attached to p.
satisfy? : (List D, Pattern R) -> Boolean
  ++ \spad{satisfy?([v1,...,vn], p)} returns \spad{f(v1,...,vn)}
++ where f is the
++ top-level predicate attached to p.
addBadValue: (Pattern R, D) -> Pattern R
  ++ addBadValue(p, v) adds v to the list of "bad values" for p;
  ++ p is not allowed to match any of its "bad values".
badValues : Pattern R -> List D
  ++ badValues(p) returns the list of "bad values" for p;
  ++ p is not allowed to match any of its "bad values".

== add
A1D ==> AnyFunctions1(D)
A1 ==> AnyFunctions1(D -> Boolean)
A1L ==> AnyFunctions1(List D -> Boolean)

applyAll: (List Any, D) -> Boolean
st : (Pattern R, List Any) -> Pattern R

st(p, l) == withPredicates(p, concat(predicates p, l))
predicate p == (d1:D):Boolean <=> applyAll(predicates p, d1)
addBadValue(p, v) == addBadValue(p, coerce(v)$A1D)
badValues p == [retract(v)$A1D for v in getBadValues p]
suchThat(p, l, f) == setTopPredicate(copy p, l, coerce(f)$A1L)
suchThat(p:Pattern R, f:D -> Boolean) == st(p, [coerce(f)$A1])
satisfy?(d:D, p:Pattern R) == applyAll(predicates p, d)
satisfy?(l:List D, p:Pattern R) ==
  empty?((rec := topPredicate p).var) => true
  retract(rec.pred)$A1L l

applyAll(l, d) ==
  for f in l repeat
    not(retract(f)$A1 d) => return false
  true

suchThat(p:Pattern R, l:List(D -> Boolean)) ==
  st(p, [coerce(f)$A1 for f in l])
package PATTERN2 PatternFunctions2

— PatternFunctions2.input —

)set break resume
)sys rm -f PatternFunctions2.output
)spool PatternFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternFunctions2
--E 1

)spool
)lisp (bye)

— PatternFunctions2.help —

====================================================================
PatternFunctions2 examples
====================================================================

Lifts maps to patterns

See Also:
  o )show PatternFunctions2
PatternFunctions2 (PATTERN2)

Exports:
map

package PATTERN2 PatternFunctions2

PatternFunctions2(R:SetCategory, S:SetCategory): with
  map: (R -> S, Pattern R) -> Pattern S
  ++ map(f, p) applies f to all the leaves of p and
  ++ returns the result as a pattern over S.
== add
  map(f, p) ==
    (r := (retractIfCan p)@Union(R, "failed")) case R =>
      f(r::R)::Pattern(S)
    (u := isOp p) case Record(op:BasicOperator, arg:List Pattern R) =>
      ur := u::Record(op:BasicOperator, arg:List Pattern R)
      (ur.op) [map(f, x) for x in ur.arg]
    (v := isQuotient p) case Record(num:Pattern R, den:Pattern R) =>
      vr := v::Record(num:Pattern R, den:Pattern R)
      map(f, vr.num) / map(f, vr.den)
    (l := isPlus p) case List(Pattern R) =>
      reduce("+", [map(f, x) for x in l::List(Pattern R)])
    (l := isTimes p) case List(Pattern R) =>
      reduce("*", [map(f, x) for x in l::List(Pattern R)])
    (x := isPower p) case
      Record(val:Pattern R, exponent: Pattern R) =>
        xr := x::Record(val:Pattern R, exponent: Pattern R)
        map(f, xr.val) ** map(f, xr.exponent)
    (w := isExpt p) case
Record(val:Pattern R, exponent: NonNegativeInteger) =>
wr := w::Record(val:Pattern R, exponent: NonNegativeInteger)
map(f, wr.val) ** wr.exponent
sy := retract(p)@Symbol
setPredicates(sy::Pattern(S), copy predicates p)

——

— PATTERN2.dotabb —

"PATTERN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATTERN2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PATTERN2" -> "FLAGG"

——

package PATMATCH PatternMatch

—— PatternMatch.input ——

)set break resume
)sys rm -f PatternMatch.output
)spool PatternMatch.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatch
--E 1

)spool
)lisp (bye)

——

— PatternMatch.help ——

====================================================================
PatternMatch examples
====================================================================

This package provides the top-level pattern matching functions.
See Also:
o  )show PatternMatch

---

PatternMatch (PATMATCH)

Exports:

is? Is

— package PATMATCH PatternMatch —

)abbrev package PATMATCH PatternMatch
++ Top-level pattern matching functions
++ Author: Manuel Bronstein
++ Date Created: 3 Dec 1989
++ Date Last Updated: 29 Jun 1990
++ Description:
++ This package provides the top-level pattern matching functions.

PatternMatch(Base, Subject, Pat): Exports == Implementation where
  Base : SetCategory
  Subject: PatternMatchable Base
  Pat : ConvertibleTo Pattern Base

Exports => with
  is?: (Subject, Pat) -> Boolean
  ++ is?(expr, pat) tests if the expression expr matches
  ++ the pattern pat.
  is?: (List Subject, Pat) -> Boolean
  ++ is?([e1,...,en], pat) tests if the list of
  ++ expressions \spad{[e1,...,en]} matches
++ the pattern pat.
Is : (List Subject, Pat) ->
    PatternMatchListResult(Base, Subject, List Subject)
++ Is([e1,...,en], pat) matches the pattern pat on the list of
++ expressions \spad{[e1,...,en]} and returns the result.
if Subject has RetractableTo(Symbol) then
    Is: (Subject, Pat) -> List Equation Subject
    ++ Is(expr, pat) matches the pattern pat on the expression
    ++ expr and returns a list of matches \spad{[v1 = e1,...,vn = en]};
    ++ returns an empty list if either expr is exactly equal to
    ++ pat or if pat does not match expr.
else
    if Subject has Ring then
        Is: (Subject, Pat) -> List Equation Polynomial Subject
        ++ Is(expr, pat) matches the pattern pat on the expression
        ++ expr and returns a list of matches \spad{[v1 = e1,...,vn = en]};
        ++ returns an empty list if either expr is exactly equal to
        ++ pat or if pat does not match expr.
    else
        Is: (Subject, Pat) -> PatternMatchResult(Base, Subject)
        ++ Is(expr, pat) matches the pattern pat on the expression
        ++ expr and returns a match of the form \spad{[v1 = e1,...,vn = en]};
        ++ returns an empty match if expr is exactly equal to pat.
        ++ returns a \spadfun{failed} match if pat does not match expr.
Implementation ==> add
import PatternMatchListAggregate(Base, Subject, List Subject)

ist: (Subject, Pat) -> PatternMatchResult(Base, Subject)

ist(s, p) == patternMatch(s, convert p, new())
is?(s: Subject, p:Pat) == not failed? ist(s, p)
is?(s:List Subject, p:Pat) == not failed? Is(s, p)
Is(s:List Subject, p:Pat) == patternMatch(s, convert p, new())

if Subject has RetractableTo(Symbol) then
    Is(s:Subject, p:Pat):List(Equation Subject) ==
    failed?(r := ist(s, p)) => empty()
    [rec.key::Subject = rec.entry for rec in destruct r]
else
    if Subject has Ring then
        Is(s:Subject, p:Pat):List(Equation Polynomial Subject) ==
        failed?(r := ist(s, p)) => empty()
        [rec.key::Polynomial(Subject) =$Equation(Polynomial Subject)
        rec.entry::Polynomial(Subject) for rec in destruct r]
    else
        Is(s:Subject,p:Pat):PatternMatchResult(Base,Subject) == ist(s,p)
package PMASS PatternMatchAssertions

---

PatternMatchAssertions.help ---

PatternMatchAssertions examples

Attaching assertions to symbols for pattern matching.
See Also:
- `)show PatternMatchAssertions`

## PatternMatchAssertions (PMASS)

![Diagram of PMASS, PID, and OAGROUP]

### Exports:
- `assert constant multiple optional`

---

```plaintext
)abbrev package PMASS PatternMatchAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching assertions to symbols for pattern matching.

PatternMatchAssertions():Exports == Implementation where
FE == Expression Integer

Exports => with
  assert : (Symbol, String) -> FE
  ++ assert(x, s) makes the assertion s about x.
  constant: Symbol -> FE
  ++ constant(x) tells the pattern matcher that x should
  ++ match only the symbol 'x and no other quantity.
  optional: Symbol -> FE
  ++ optional(x) tells the pattern matcher that x can match
  ++ an identity (0 in a sum, 1 in a product or exponentiation).
  multiple: Symbol -> FE
  ++ multiple(x) tells the pattern matcher that x should
```

---
++ preferably match a multi-term quantity in a sum or product.
++ For matching on lists, multiple(x) tells the pattern matcher
++ that x should match a list instead of an element of a list.

Implementation ==> add
import FunctionSpaceAssertions(Integer, FE)

constant x == constant(x::FE)
multiple x == multiple(x::FE)
optional x == optional(x::FE)
assert(x, s) == assert(x::FE, s)

— PMASS.dotabb —

"PMASS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMASS"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PMASS" -> "PID"
"PMASS" -> "OAGROUP"

package PMFS PatternMatchFunctionSpace

— PatternMatchFunctionSpace.input —

)set break resume
)sys rm -f PatternMatchFunctionSpace.output
)spool PatternMatchFunctionSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchFunctionSpace
--E 1

)spool
)lisp (bye)
This package provides pattern matching functions on function spaces.

See Also:
s )show PatternMatchFunctionSpace

PatternMatchFunctionSpace (PMFS)

Exports:
patternMatch

— package PMFS PatternMatchFunctionSpace —

)abbrev package PMFS PatternMatchFunctionSpace
++ Author: Manuel Bronstein
++ Date Created: 15 Mar 1990
++ Date Last Updated: 20 June 1991
++ Description:
++ This package provides pattern matching functions on function spaces.

PatternMatchFunctionSpace(S, R, F): Exports== Implementation where
  S: SetCategory
  R: Join(IntegralDomain, OrderedSet, PatternMatchable S)
  F: Join(FunctionSpace R,ConvertibleTo Pattern S, PatternMatchable S,
                        RetractableTo Kernel %) -- that one is redundant but won't
-- compile without it

N ==> NonNegativeInteger
K ==> Kernel F
PAT ==> Pattern S
PRS ==> PatternMatchResult(S, F)
RCP ==> Record(val: PAT, exponent: N)
RCX ==> Record(var: K, exponent: Integer)

Exports ==> with
    patternMatch: (F, PAT, PRS) -> PRS
    ++ patternMatch(expr, pat, res) matches the pattern pat to the
    ++ expression expr; res contains the variables of pat which
    ++ are already matched and their matches.

Implementation ==> add
    import PatternMatchKernel(S, F)
    import PatternMatchTools(S, R, F)
    import PatternMatchPushDown(S, R, F)

    patternMatch(x, p, 1) ==
        generic? p => addMatch(p, x, 1)
        (r := retractIfCan(x)@Union(R, "failed")) case R =>
            patternMatch(r::R, p, 1)
        (v := retractIfCan(x)@Union(K, "failed")) case K =>
            patternMatch(v::K, p, 1)
        (q := isQuotient p) case Record(num: PAT, den: PAT) =>
            uq := q::Record(num: PAT, den: PAT)
            failed?(l := patternMatch(numer(x):: F, uq.num, l)) => l
            patternMatch(denom(x):: F, uq.den, l)
        (u := isPlus p) case List(PAT) =>
            (lx := isPlus x) case List(F) =>
                patternMatch(lx:: List(F), u:: List(PAT), l1 +-> +/l1, l, patternMatch)
            (u := optpair(u:: List(PAT))) case List(PAT) =>
                failed?(l := addMatch(first(u:: List(PAT)), 0, l)) => failed()
                patternMatch(x, second(u:: List(PAT)), l)
            failed()
        (u := isTimes p) case List(PAT) =>
            (lx := isTimes x) case List(F) =>
                patternMatchTimes(lx:: List(F), u:: List(PAT), l, patternMatch)
            (u := optpair(u:: List(PAT))) case List(PAT) =>
                failed?(l := addMatch(first(u:: List(PAT)), 1, l)) => failed()
                patternMatch(x, second(u:: List(PAT)), l)
            failed()
        (uu := isPower p) case Record(val: PAT, exponent: PAT) =>
            uur := uu::Record(val: PAT, exponent: PAT)
        (ex := isExpt x) case RCX =>
            failed?(l := patternMatch((ex::RCX).exponent:: Integer:: F,
                uur.exponent, l)) => failed()
            patternMatch((ex::RCX).var, uur.val, l)
optional?(uur.exponent) =>
  failed?(l := addMatch(uur.exponent, 1, l)) => failed()
patternMatch(x, uur.val, 1)
failed()
((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
(ex::RCX).exponent = ((ep::RCP).exponent)::Integer =>
  patternMatch((ex::RCX).var, (ep::RCP).val, 1)
failed()
This package provides pattern matching functions on integers.

See Also:
- \texttt{\textbackslash )show PatternMatchIntegerNumberSystem}

---

\textbf{PatternMatchIntegerNumberSystem (PMINS)}

\texttt{PMINS}

\texttt{FLAGG}\hspace{1cm}\texttt{FLAGG-}

\textbf{Exports:}

\begin{itemize}
  \item \texttt{patternMatch}
\end{itemize}

\begin{verbatim}
)abbrev package PMINS PatternMatchIntegerNumberSystem
++ Author: Manuel Bronstein
++ Date Created: 29 Nov 1989
++ Date Last Updated: 22 Mar 1990
++ Description:
++ This package provides pattern matching functions on integers.

PatternMatchIntegerNumberSystem(I:IntegerNumberSystem): with
    patternMatch: (I, Pattern Integer, PatternMatchResult(Integer, I)) ->
        PatternMatchResult(Integer, I)
    ++ patternMatch(n, pat, res) matches the pattern pat to the
    ++ integer n; res contains the variables of pat which
    ++ are already matched and their matches.
    ++
    == add
    import IntegerRoots(I)

    PAT => Pattern Integer
\end{verbatim}

---
CHAPTER 17. CHAPTER P

PMR ==> PatternMatchResult(Integer, I)

patternMatchInner : (I, PAT, PMR) -> PMR
patternMatchRestricted: (I, PAT, PMR, I) -> PMR
patternMatchSumProd : (I, List PAT, PMR, (I, I) -> Union(I, "failed"), I) -> PMR

patternMatch(x, p, l) ==
  generic? p => addMatch(p, x, l)
  patternMatchInner(x, p, l)

patternMatchRestricted(x, p, l, y) ==
  generic? p => addMatchRestricted(p, x, l, y)
  patternMatchInner(x, p, l)

patternMatchSumProd(x, lp, l, invOp, ident) ==
  #lp = 2 =>
  p2 := last lp
  if ((r := retractIfCan(p1 := first lp)@Union(Integer,"failed"))
    case "failed" then (p1 := p2; p2 := first lp)
    (r := retractIfCan(p1)@Union(Integer, "failed")) case "failed" =>
      failed()
    (y := invOp(x, r::Integer::I)) case "failed" => failed()
    patternMatchRestricted(y::I, p2, l, ident)
    failed()

patternMatchInner(x, p, l) ==
  constant? p =>
    (r := retractIfCan(p)@Union(Integer, "failed")) case Integer =>
      convert(x)@Integer = r::Integer => l
      failed()
    failed()
  (u := isExpt p) case Record(val:PAT,exponent:NonNegativeInteger) =>
    ur := u::Record(val:PAT, exponent:NonNegativeInteger)
    (v := perfectNthRoot(x, ur.exponent)) case "failed" => failed()
    patternMatchRestricted(v::I, ur.val, l, 1)
  (uu := isPower p) case Record(val:PAT, exponent:PAT) =>
    uur := uu::Record(val:PAT, exponent: PAT)
    pr := perfectNthRoot x
    failed?(l := patternMatchRestricted(pr.exponent::Integer::I,
      uur.exponent, l, 1)) => failed()
    patternMatchRestricted(pr.base, uur.val, l, 1)
  (w := isTimes p) case List(PAT) =>
    patternMatchSumProd(x, w::List(PAT), l,
      (i1:I,i2:I):Union(I, "failed") +-> i1 exquo i2, 1)
  (w := isPlus p) case List(PAT) =>
    patternMatchSumProd(x,w::List(PAT),l,
      (i1:I,i2:I):Union(I,"failed") +-> (i1-i2)::Union(I,"failed"),0)
  (uv := isQuotient p) case Record(num:PAT, den:PAT) =>
    uvr := uv::Record(num:PAT, den:PAT)
(r := retractIfCan(uvr.num)@Union(Integer,"failed")) case Integer
and (v := r::Integer::I exquo x) case I =>
  patternMatchRestricted(v::I, uvr.den, 1, 1)
(r := retractIfCan(uvr.den)@Union(Integer,"failed")) case Integer
  => patternMatch(r::Integer * x, uvr.num, l)
failed()
failed()

package INTPM PatternMatchIntegration

— PatternMatchIntegration.input —

)set break resume
)sys rm -f PatternMatchIntegration.output
)spool PatternMatchIntegration.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PatternMatchIntegration
--E 1

)spool
)lisp (bye)

— PatternMatchIntegration.help —

====================================================================
PatternMatchIntegration examples
====================================================================

PatternMatchIntegration provides functions that use the pattern matcher to find some indefinite and definite integrals involving special functions and found in the litterature.

See Also:
  o )show PatternMatchIntegration

----------

PatternMatchIntegration (INTPM)

Exports:
  pmComplexIntegrate  pmIntegrate  pmIntegrate  splitConstant

--- package INTPM PatternMatchIntegration ---

)abbrev package INTPM PatternMatchIntegration
++ Author: Manuel Bronstein
++ Date Created: 5 May 1992
++ Date Last Updated: 27 September 1995
++ Description:
++ \spadtype{PatternMatchIntegration} provides functions that use
++ the pattern matcher to find some indefinite and definite integrals
++ involving special functions and found in the litterature.

PatternMatchIntegration(R, F): Exports == Implementation where
  R : Join(OrderedSet, RetractableTo Integer, GcdDomain,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)
\textbf{PACKAGE INTPM PATTERNMATCHINTEGRATION}

\textbf{N} \implies \text{NonNegativeInteger}
\textbf{Z} \implies \text{Integer}
\textbf{SY} \implies \text{Symbol}
\textbf{K} \implies \text{Kernel F}
\textbf{P} \implies \text{SparseMultivariatePolynomial}(R, K)
\textbf{SUP} \implies \text{SparseUnivariatePolynomial} F
\textbf{PAT} \implies \text{Pattern} Z
\textbf{RES} \implies \text{PatternMatchResult}(Z, F)
\textbf{OFE} \implies \text{OrderedCompletion} F
\textbf{REC} \implies \text{Record}(which: Z, exponent: F, coeff: F)
\textbf{ANS} \implies \text{Record}(special:F, integrand:F)
\textbf{NONE} \implies 0
\textbf{EI} \implies 1
\textbf{ERF} \implies 2
\textbf{SI} \implies 3
\textbf{CI} \implies 4
\textbf{GAM2} \implies 5
\textbf{CI0} \implies 6

\textbf{Exports} \implies \text{with}
\textbf{splitConstant}: (F, SY) \rightarrow \text{Record}(\text{const:F}, \text{nconst:F})
++ \text{splitConstant}(f, x) \text{ returns } \text{spad\{c, g\} such that}
++ \text{spad\{f = c * g\} and spad\{c\} does not involve spad\{t\}.}

\text{if R has ConvertibleTo Pattern Integer and}
\text{R has PatternMatchable Integer then}
\text{if F has LiouvillianFunctionCategory then}
\text{pmComplexintegrate}: (F, SY) \rightarrow \text{Union(ANS, "failed")}
++ \text{pmComplexintegrate}(f, x) \text{ returns either "failed" or}
++ \text{spad\{g,h\} such that}
++ \text{spad\{\text{integrate}(f,x) = g + integrate(h,x)\}.}
++ \text{It only looks for special complex integrals that pmintegrate}
++ \text{does not return.}

\text{pmintegrate}: (F, SY) \rightarrow \text{Union(ANS, "failed")}
++ \text{pmintegrate}(f, x) \text{ returns either "failed" or spad\{g,h\} such}
++ \text{that spad\{\text{integrate}(f,x) = g + integrate(h,x)\}.}

\text{if F has SpecialFunctionCategory then}
\text{pmintegrate}: (F, SY, OFE, OFE) \rightarrow \text{Union(F, "failed")}
++ \text{pmintegrate}(f, x = a..b) \text{ returns the integral of}
++ \text{spad\{f(x)dx\} from a to b}
++ \text{if it can be found by the built-in pattern matching rules.}

\textbf{Implementation} \implies \text{add}
\text{import PatternMatch(Z, F, F)}
\text{import ElementaryFunctionSign(R, F)}
\text{import FunctionSpaceAssertions(R, F)}
\text{import TrigonometricManipulations(R, F)}
\text{import FunctionSpaceAttachPredicates(R, F, F)}

\text{mkalist} : \text{RES} \rightarrow \text{AssociationList(SY, F)}
pm := new()$SY
pmw := new pm
pmm := new pm
pms := new pm
pmc := new pm
pma := new pm
pmb := new pm
c := optional(pmc::F)
w := suchThat(optional(pmw::F),
  (x1:F):Boolean +-> empty? variables x1)
s := suchThat(optional(pms::F),
  (x1:F):Boolean +-> empty? variables x1 and real? x1)
m := suchThat(optional(pmm::F),
  (x1:F):Boolean+->(retractIfCan(x1)@Union(Z,"failed") case Z) and x1 >= 0)
spi := sqrt(pi()$F)

half := 1::F / 2::F

mkalist res == construct destruct res

splitConstant(f, x) ==
  not member?(x, variables f) => [f, 1]
  (retractIfCan(f)@Union(K, "failed")) case K => [1, f]
  (u := isTimes f) case List(F) =>
    cc := nc := 1$F
    for g in u::List(F) repeat
      rec := splitConstant(g, x)
      cc := cc * rec.const
      nc := nc * rec.nconst
    [cc, nc]
  (u := isPlus f) case List(F) =>
    rec := splitConstant(first(u::List(F)), x)
    cc := rec.const
    nc := rec.nconst
    for g in rest(u::List(F)) repeat
      rec := splitConstant(g, x)
      if rec.nconst = nc then cc := cc + rec.const
      else if rec.nconst = -nc then cc := cc - rec.const
      else return [1, f]
    [cc, nc]
  if (v := isPower f) case Record(val:F, exponent:Z) then
    vv := v::Record(val:F, exponent:Z)
    (vv.exponent "= 1") =>
      rec := splitConstant(vv.val, x)
      return [rec.const ** vv.exponent, rec.nconst ** vv.exponent]
  error "splitConstant: should not happen"
if $R$ has ConvertibleTo Pattern Integer and $R$ has PatternMatchable Integer then
if $F$ has LiouvillianFunctionCategory then
import ElementaryFunctionSign($R$, $F$)

\[
\begin{align*}
\text{insqrt} & : F \to F \\
\text{matchei} & : (F, SY) \to \text{REC} \\
\text{matchfer} & : (F, SY, Boolean) \to \text{REC} \\
\text{matchsi} & : (F, SY) \to \text{REC} \\
\text{matchli} & : (F, SY) \to \text{List F} \\
\text{matchli0} & : (F, K, SY) \to \text{List F} \\
\text{matchdilog} & : (F, SY) \to \text{List F} \\
\text{matchdilog0} & : (F, K, SY, P, F) \to \text{List F} \\
\text{goodlilog} & : (K, P) \to \text{Boolean} \\
\text{gooddilog} & : (K, P, F) \to \text{Boolean}
\end{align*}
\]

-- gooddilog?(k, p) == is?(k, "log":SY) and one? minimumDegree(p, k)

\[
\begin{align*}
\text{gooddilog} & : (K, P) \to \text{Boolean} \\
\text{gooddilog}?(k, p) & == \text{is?(k, "log":SY) and (minimumDegree(p, k) = 1)}
\end{align*}
\]

-- matches the integral to a result of the form $d \ast \text{erf}(u)$ or $d \ast \text{ei}(u)$
-- returns $\text{[case, u, d]}$

\[
\begin{align*}
\text{matchfer}(f, x, \text{comp?}) & == \\
\text{res0} & := \text{new()}\text{RES} \\
\text{pat} & := c \ast \exp(pma::F) \\
\text{failed?}(\text{res} := \text{patternMatch}(f, \text{convert(pat)}@\text{PAT}, \text{res0})) & => \\
\text{comp?} & => \text{[NONE, 0, 0]} \\
\text{matchei}(f, x) & \\
\text{l} & := \text{mkalist res} \\
\text{da} & := \text{differentiate}(a := l.pma, x) \\
\text{d} & := a \ast (cc := l.pmc) / \text{da} \\
\text{zero? differentiative}(d, x) & => \text{[EI, a, d]} \\
\text{comp? or ((u := sign a) case Z) and (u::Z) < 0) =>} \\
\text{d} & := cc \ast (sa := \text{insqrt}(- a)) / \text{da} \\
\text{zero? differentiative}(d, x) & => \text{[ERF, sa, - d * spi]} \\
\text{[NONE, 0, 0]} \\
\text{[NONE, 0, 0]}
\end{align*}
\]

-- matches the integral to a result of the form $d \ast \text{ei}(k \ast \text{log} u)$
-- returns $\text{[case, k * log u, d]}$

\[
\begin{align*}
\text{matchei}(f, x) & == \\
\text{res0} & := \text{new()}\text{RES} \\
\text{a} & := \text{pma::F} \\
\text{pat} & := c \ast a**w / \text{log a} \\
\text{failed?}(\text{res} := \text{patternMatch}(f, \text{convert(pat)}@\text{PAT}, \text{res0})) & => \\
\text{[NONE, 0, 0]} \\
\text{l} & := \text{mkalist res}
\end{align*}
\]
da := differentiate(a := l.pma, x)
d := (cc := l.pmc) / da
zero? differentiate(d, x) => [EI, (1 + l.pmw) * log a, d]
[NONE, 0, 0]

-- matches the integral to a result of the form d * dilog(u) + int(v),
-- returns [u,d,v] or []
machdidilog(f, x) ==
n := numer f
df := (d := denom f)::F
for k in select_!(
  (x1:K):Boolean +-> gooddilog?(x1,n,d),variables n)$List(K) repeat
  not empty?(l := machdidilog0(f, k, x, n, df)) => return l
empty()

-- matches the integral to a result of the form d * dilog(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
machdilog0(f, k, x, p, q) ==
  zero?(da := differentiate(a := first argument k, x)) => empty()
a1 := 1 - a
d := coefficient(univariate(p, k), 1)::F * a1 / (q * da)
  zero? differentiate(d, x) => [a, d, f - d * da * (k::F) / a1]
empty()

-- matches the integral to a result of the form d * li(u) + int(v),
-- returns [u,d,v] or []
machli(f, x) ==
d := denom f
for k in select_!(
  (x1:K):Boolean +-> goodlilog?(x1,d),variables d)$List(K) repeat
  not empty?(l := machli0(f, k, x)) => return l
empty()

-- matches the integral to a result of the form d * li(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
machli0(f, k, x) ==
g := (lg := k::F) * f
  zero?(da := differentiate(a := first argument k, x)) => empty()
  zero? differentiate(d := g / da, x) => [a, d, 0]
ug := univariate(g, k)
(u:=retractIfCan(ug)@Union(SUP,"failed")) case "failed" => empty()
degree(p := u::SUP) > 1 => empty()
  zero? differentiate(d := coefficient(p, 0) / da, x) =>
    [a, d, leadingCoefficient p]
empty()

-- matches the integral to a result of the form d * Si(u) or d * Ci(u)
-- returns [case, u, d]
matchsici(f, x) ==
  res0 := new()$RES
  b := pmb::F
  t := tan(a := pma::F)
  patsi := c * t / (patden := b + b * t**2)
  patci := (c - c * t**2) / patden
  patci0 := c / patden

  ci0?:Boolean
  (ci? := failed?(res := patternMatch(f, convert(patsi)@PAT, res0)))
  and (ci0?:=failed?(res:=patternMatch(f,convert(patci)@PAT,res0)))
  and failed?(res := patternMatch(f,convert(patci0)@PAT,res0)) =>
    [NONE, 0, 0]

  l := mkalist res
  (b := l.pmb) ^= 2 * (a := l.pma) => [NONE, 0, 0]
  db := differentiate(b, x)
  d := (cc := l.pmc) / db
  zero? differentiate(d, x) =>
    ci? =>
      ci0? => [CI0, b, d / (2::F)]
      [CI, b, d]
      [SI, b, d / (2::F)]
    [NONE, 0, 0]

-- returns a simplified sqrt(y)
insqrt y ==
  rec := froot(y, 2)$PolynomialRoots(IndexedExponents K, K, R, P, F)
  ((rec.exponent) = 1) => rec.coef * rec.radicand
  rec.exponent ^=2 => error "insqrt: hould not happen"
  rec.coef * sqrt(rec.radicand)

pmintegrate(f, x) ==
  (rc := splitConstant(f, x)).const ^= 1 =>
    (u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
    rec := u::ANS
    [rc.const * rec.special, rc.const * rec.integrand]
  not empty?(l := matchli(f, x)) => [second l * li first l, third l]
  not empty?(l := matchdilog(f, x)) =>
    [second l * dilog first l, third l]
  cse := (rec := matcherfei(f, x, false)).which
  cse = EI => [rec.coef * Ei(rec.exponent), 0]
  cse = ERF => [rec.coef * erf(rec.exponent), 0]
  cse := (rec := matchsici(f, x)).which
  cse = SI => [rec.coef * Si(rec.exponent), 0]
  cse = CI => [rec.coef * Ci(rec.exponent), 0]
  cse = CI0 => [rec.coef * Ci(rec.exponent) + rec.coef * log(rec.exponent), 0]
  "failed"

pmComplexintegrate(f, x) ==
(rc := splitConstant(f, x)).const ^= 1 =>
(u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
rec := u::ANS
[rc.const * rec.special, rc.const * rec.integrand]
cse := (rec := matcherfei(f, x, true)).which
cse = ERF => [rec.coef * erf(rec.coefficient), 0]
"failed"

if F has SpecialFunctionCategory then
match1 : (F, SY, F, F) -> List F
formulal : (F, SY, F, F) -> Union(F, "failed")

-- tries only formula (1) of the Geddes & al, AAECC 1 (1990) paper
formulal(f, x, t, cc) ==
empty?(l := match1(f, x, t, cc)) => "failed"
mw := first l
zero?(ms := third l) or ((sgs := sign ms) case "failed")=> "failed"
((sgz := sign(z := (mw + 1) / ms)) case "failed") or (sgz::Z < 0)
=> "failed"
mmi := retract(mm := second l)@Z
sgs * (last l) * ms**(- mmi - 1) * 
eval(differentiate(Gamma(x::F), x, mmi::N), [kernel(x)@K], [z])

-- returns [w, m, s, c] or []
-- matches only formula (1) of the Geddes & al, AAECC 1 (1990) paper
match1(f, x, t, cc) ==
res0 := new()$RES
pat := cc * log(t)**m * exp(-t**s)
not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
1 := mkalist res
[0, 1.pmm, 1.pms, 1.pmc]
pat := cc * t**w * exp(-t**s)
not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
1 := mkalist res
[1.pmw, 0, 1.pms, 1.pmc]
pat := cc / t**w * exp(-t**s)
not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
1 := mkalist res
[- 1.pmw, 0, 1.pms, 1.pmc]
pat := cc * t**w * log(t)**m * exp(-t**s)
not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
1 := mkalist res
[1.pmw, 1.pmm, 1.pms, 1.pmc]
pat := cc / t**w * log(t)**m * exp(-t**s)
not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
1 := mkalist res
[- 1.pmw, 1.pmm, 1.pms, 1.pmc]
empty()

pmintegrate(f, x, a, b) ==
--
zero? a and one? whatInfinity b =>
zero? a and ((whatInfinity b) = 1) =>
formula1(f, x, constant(x::F),
suchThat(c, (x1:F):Boolean +-> freeOf?(x1, x)))
"failed"

---

-- INTPM.dotabb --

"INTPM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPM"]
"ACF" [color="#FF4488",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#FF4488",href="bookvol10.2.pdf#nameddest=FS"]
"INTPM" -> "ACF"
"INTPM" -> "FS"

---

package PMKERNEL PatternMatchKernel

--- PatternMatchKernel.input ---

`}set break resume
`sys rm -f PatternMatchKernel.output
`spool PatternMatchKernel.output
`set message test on
`set message auto off
`clear all

|--S 1 of 1
`show PatternMatchKernel
|--E 1

`spool
`lisp (bye)

---

--- PatternMatchKernel.help ---

====================================================================
PatternMatchKernel examples
====================================================================
This package provides pattern matching functions on kernels.

See Also:
- `show PatternMatchKernel`

---

PatternMatchKernel (PMKERNEL)

Exports:
- `patternMatch`

--- package PMKERNEL PatternMatchKernel ---

`abbrev package PMKERNEL PatternMatchKernel`  
++ Author: Manuel Bronstein  
++ Date Created: 12 Jan 1990  
++ Date Last Updated: 4 May 1992  
++ Description:  
++ This package provides pattern matching functions on kernels.

PatternMatchKernel(S, E): Exports == Implementation where  
S: SetCategory  
E: Join(OrderedSet, RetractableTo Kernel %,  
    ConvertibleTo Pattern S, PatternMatchable S)

PAT => Pattern S  
PRS => PatternMatchResult(S, E)  
POWER => "%power":Symbol  
NTHRT => "nthRoot":Symbol

Exports => with
patternMatch: (Kernel E, PAT, PRS) -> PRS
  ++ \texttt{patternMatch}(f(e_1, \ldots, e_n), \texttt{pat}, \texttt{res}) matches the pattern \texttt{pat}
  ++ to \texttt{\verb|\spad{f(e_1, \ldots, e_n)|}; \texttt{res} contains the variables of \texttt{pat} which
  ++ are already matched and their matches.

Implementation ==> add

\texttt{patternMatchArg} : (List E, List PAT, PRS) -> PRS
\texttt{patternMatchInner}: (Kernel E, PAT, PRS) -> \texttt{Union}(PRS, "failed")

-- matches the ordered lists \texttt{ls} and \texttt{lp}.
\texttt{patternMatchArg}(ls, lp, l) ==
  \#ls ^= \#lp => \texttt{failed()}
  for p in lp for s in ls repeat
    generic? p and failed?(l := addMatch(p, s, l)) => return \texttt{failed()}
  for p in lp for s in ls repeat
    not(generic? p) and failed?(l := \texttt{patternMatch}(s, p, l)) =>
      return \texttt{failed()}
  l

\texttt{patternMatchInner}(s, p, l) ==
  generic? p => \texttt{addMatch}(p, s::E, l)
  (u := isOp p) case Record(op:BasicOperator, arg: List PAT) =>
    ur := u::Record(op:BasicOperator, arg: List PAT)
    ur.op = operator s => \texttt{patternMatchArg(argument s, ur.arg, l)}
    failed()
  constant? p =>
    ((v := retractIfCan(p)@Union(Symbol, "failed")) case Symbol) and
    ((w := symbolIfCan s) case Symbol) and
    (v::Symbol = w::Symbol) => l
    \texttt{failed()}
  "failed"

if \texttt{E} has Monoid then
\texttt{patternMatchMonoid}: (Kernel E, PAT, PRS) -> \texttt{Union}(PRS, "failed")
\texttt{patternMatchOpt} : (E, List PAT, PRS, E) -> PRS

\texttt{patternMatchOpt}(x, lp, l, id) ==
  (u := optpair lp) case List(PAT) =>
    failed?(l := addMatch(first(u::List(PAT)), id, l)) => \texttt{failed()}
    \texttt{patternMatch(x, second(u::List(PAT)), l)}
    failed()

\texttt{patternMatchMonoid}(s, p, l) ==
  (u := \texttt{patternMatchInner}(s, p, l)) case PRS => u::PRS
  (v := isPower p) case Record(val:PAT, exponent:PAT) =>
    vr := v::Record(val:PAT, exponent: PAT)
    is?(op := operator s, POWER) =>
      \texttt{patternMatchArg(argument s, [vr.val, vr.exponent], l)}
    is?(op,NTHRT) and ((r := recip(second(arg := argument s))) case E) =>
      \texttt{patternMatchArg([first arg, r::E], [vr.val, vr.exponent], l)}
optional?(vr.exponent) =>
  failed?(l := addMatch(vr.exponent, l, 1)) => failed()
  patternMatch(s::E, vr.val, l)
  failed()

  (w := isTimes p) case List(PAT) =>
    patternMatchOpt(s::E, w::List(PAT), 1, 1)
    "failed"

if E has AbelianMonoid then
  patternMatch(s, p, l) ==
    (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
    (w := isPlus p) case List(PAT) =>
      patternMatchOpt(s::E, w::List(PAT), 1, 0)
      failed()
  else
    patternMatch(s, p, l) ==
      (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
      failed()
  else
    patternMatch(s, p, l) ==
      (u := patternMatchInner(s, p, l)) case PRS => u::PRS
      failed()

— PMKERNEL.dotabb —

"PMKERNEL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMKERNEL"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PMKERNEL" -> "ALIST"

package PMLSAGG PatternMatchListAggregate

— PatternMatchListAggregate.input —

)set break resume
)sys rm -f PatternMatchListAggregate.output
)spool PatternMatchListAggregate.output
)set message test on
)set message auto off
This package provides pattern matching functions on lists.

See Also:
  o )show PatternMatchListAggregate

---

PatternMatchListAggregate (PMLSAGG)

Exports:
  patternMatch

---

)abbrev package PMLSAGG PatternMatchListAggregate

)clear all

--S 1 of 1
)show PatternMatchListAggregate
--E 1

)spool
)lisp (bye)

---

— PatternMatchListAggregate.help —

This package provides pattern matching functions on lists.

See Also:
  o )show PatternMatchListAggregate
This package provides pattern matching functions on lists.

PatternMatchListAggregate(S, R, L): Exports == Implementation where
S: SetCategory
R: PatternMatchable S
L: ListAggregate R

PLR => PatternMatchListResult(S, R, L)

Exports => with
patternMatch: (L, Pattern S, PLR) -> PLR
++ patternMatch(l, pat, res) matches the pattern pat to the
++ list l; res contains the variables of pat which
++ are already matched and their matches.

Implementation => add
match: (L, List Pattern S, PLR, Boolean) -> PLR

patternMatch(l, p, r) ==
(u := isList p) case "failed" => failed()
match(l, u::List Pattern S, r, true)

match(l, lp, r, new?) ==
empty? lp =>
  empty? l => r
  failed()
multiple?(p0 := first lp) =>
  empty? rest lp =>
    if not new? then l := reverse_! l
    makeResult(atoms r, addMatchRestricted(p0,l,lists r,empty()))
  new? => match(reverse l, reverse lp, r, false)
  error "Only one multiple pattern allowed in list"
empty? l => failed()
failed?(r := makeResult(patternMatch(first l,p0,atoms r),lists r)) => failed()
match(rest l, rest lp, r, new?)

— PMLSAGG.dotabb —

"PMLSAGG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMLSAGG"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMLSAGG" -> "FLAGG"
package PMPLCAT PatternMatchPolynomialCategory

---

PatternMatchPolynomialCategory.input ---

)set break resume
/sys rm -f PatternMatchPolynomialCategory.output
/spool PatternMatchPolynomialCategory.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchPolynomialCategory
--E 1

)spool
)lisp (bye)

---

PatternMatchPolynomialCategory.help ---

====================================================================
PatternMatchPolynomialCategory examples
====================================================================

This package provides pattern matching functions on polynomials.

See Also:
o )show PatternMatchPolynomialCategory
PatternMatchPolynomialCategory (PMPLCAT)

Exports:
 patternMatch

-- package PMPLCAT PatternMatchPolynomialCategory --

abbrev package PMPLCAT PatternMatchPolynomialCategory
 ++ Author: Manuel Bronstein
 ++ Date Created: 9 Jan 1990
 ++ Date Last Updated: 20 June 1991
 ++ Description:
 ++ This package provides pattern matching functions on polynomials.

PatternMatchPolynomialCategory(S,E,V,R,P):Exports== Implementation where
 S: SetCategory
 E: OrderedAbelianMonoidSup
 V: OrderedSet
 R: Join(Ring, OrderedSet, PatternMatchable S)
 P: Join(PolynomialCategory(R, E, V), ConvertibleTo Pattern S)

 N --> NonNegativeInteger
 PAT --> Pattern S
 PRS --> PatternMatchResult(S, P)
 RCP --> Record(val:PAT, exponent:N)
 RCX --> Record(var:V, exponent:N)

Exports --> with
 patternMatch: (P, PAT, PRS, (V, PAT, PRS) --> PRS) --> PRS
  ++ patternMatch(p, pat, res, vmatch) matches the pattern pat to
  ++ the polynomial p. res contains the variables of pat which
  ++ are already matched and their matches; vmatch is the matching
  ++ function to use on the variables.
  -- This can be more efficient than pushing down when the variables
  -- are recursive over P (e.g. kernels)
 if V has PatternMatchable S then
  patternMatch: (P, PAT, PRS) --> PRS
patternMatch(p, pat, res) matches the pattern pat to
the polynomial p; res contains the variables of pat which
are already matched and their matches.

Implementation ==> add
import PatternMatchTools(S, R, P)
import PatternMatchPushDown(S, R, P)

if V has PatternMatchable S then
  patternMatch(x, p, l) ==
    patternMatch(x, p, l, patternMatch$PatternMatchPushDown(S,V,P))

patternMatch(x, p, l, vmatch) ==
generic? p => addMatch(p, x, l)
(r := retractIfCan(x)@Union(R, "failed")) case R =>
  patternMatch(r::R, p, l)
(v := retractIfCan(x)@Union(V, "failed")) case V =>
  vmatch(v::V, p, l)
(u := isPlus p) case List(PAT) =>
  (lx := isPlus x) case List(P) =>
    patternMatch(lx::List(P), u::List(PAT),
      (l1:List(P)):P +-> */l1, l,
(u := optpair(u::List(PAT))) case List(PAT) =>
  failed?(l := addMatch(first(u::List(PAT)), 0, l)) => failed()
  patternMatch(x, second(u::List(PAT)), l, vmatch)
  failed()
(u := isTimes p) case List(PAT) =>
  (lx := isTimes x) case List(P) =>
    patternMatchTimes(lx::List(P), u::List(PAT), l,
(u := optpair(u::List(PAT))) case List(PAT) =>
  failed?(l := addMatch(first(u::List(PAT)), 1, l)) => failed()
  patternMatch(x, second(u::List(PAT)), l, vmatch)
  failed()
(uu := isPower p) case Record(val:PAT, exponent:PAT) =>
  uur := uu::Record(val:PAT, exponent: PAT)
  (ex := isExpt x) case RCX =>
    failed?(l := patternMatch((ex::RCX).exponent::Integer::P,
      uur.exponent, l, vmatch)) => failed()
    vmatch((ex::RCX).var, uur.val, l)
  optional?(uur.exponent) =>
    failed?(l := addMatch(uur.exponent, 1, l)) => failed()
    patternMatch(x, uur.val, l, vmatch)
    failed()
((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
(ex::RCX).exponent = (ep::RCP).exponent =>
  vmatch((ex::RCX).var, (ep::RCP).val, 1)
failed()
package PMDOWN PatternMatchPushDown

-- PatternMatchPushDown.input --

)set break resume
)sys rm -f PatternMatchPushDown.output
)spool PatternMatchPushDown.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchPushDown
--E 1

)spool
)lisp (bye)

-- PatternMatchPushDown.help --

====================================================================
PatternMatchPushDown examples
====================================================================

This packages provides tools for matching recursively in type towers.

See Also:
o )show PatternMatchPushDown

--
PatternMatchPushDown (PMDOWN)

Exports:
fixPredicate  patternMatch

— package PMDOWN PatternMatchPushDown —

)abbrev package PMDOWN PatternMatchPushDown
++ Author:  Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 16 August 1995
++ Description:
++ This packages provides tools for matching recursively in type towers.

PatternMatchPushDown(S, A, B): Exports == Implementation where
  S: SetCategory
  A: PatternMatchable S
  B: Join(SetCategory, RetractableTo A)

  PAT ==> Pattern S
  PRA ==> PatternMatchResult(S, A)
  PRB ==> PatternMatchResult(S, B)
  REC ==> Record(pat:PAT, res:PRA)

Exports ==> with
  fixPredicate: (B -> Boolean) -> (A -> Boolean)
    ++ fixPredicate(f) returns g defined by g(a) = f(a::B);
  patternMatch: (A, PAT, PRB) -> PRB
    ++ patternMatch(expr, pat, res) matches the pattern pat to the
      ++ expression expr; res contains the variables of pat which
      ++ are already matched and their matches.
    ++ Note that this function handles type towers by changing the predicates
      ++ and calling the matching function provided by \spad{A}.

Implementation ==> add
  import PatternMatchResultFunctions2(S, A, B)
fixPred : Any -> Union(Any, "failed")
inA : (PAT, PRB) -> Union(List A, "failed")
fixPredicates: (PAT, PRB, PRA) -> Union(REC, "failed")
fixList: (List PAT -> PAT, List PAT, PRB, PRA) -> Union(REC,"failed")

fixPredicate f == (a1:A):Boolean +-> f(a1::B)

patternMatch(a, p, l) ==
  (u := fixPredicates(p, l, new())) case "failed" => failed()
  union(l, map((a1:A):B +->a1::B,
  patternMatch(a, (u::REC).pat, (u::REC).res)))

inA(p, l) ==
  (u := getMatch(p, l)) case "failed" => empty()
  (r := retractIfCan(u::B)@Union(A, "failed")) case A => [r::A]
  "failed"

fixList(fn, l, lb, la) ==
  ll:List(PAT) := empty()
  for x in l repeat
    (f := fixPredicates(x, lb, la)) case "failed" => return "failed"
    ll := concat((f::REC).pat, ll)
    la := (f::REC).res
  [fn ll, la]

fixPred f ==
  (u:= retractIfCan(f)$AnyFunctions1(B -> Boolean)) case "failed" => "failed"
  g := fixPredicate(u::(B -> Boolean))
  coerce(g)$AnyFunctions1(A -> Boolean)

fixPredicates(p, lb, la) ==
  (r:=retractIfCan(p)@Union(S,"failed")) case S or quoted? p =>[p,la]
  (u := isOp p) case Record(op:BasicOperator, arg:List PAT) =>
    ur := u::Record(op:BasicOperator, arg:List PAT)
    fixList((ll:List(PAT)):PAT+-> (ur.op) ll, ur.arg, lb, la)
  (us := isPlus p) case List(PAT) =>
    fixList((ll:List(PAT)):PAT+->reduce("+", ll), us::List(PAT), lb, la)
  (us := isTimes p) case List(PAT) =>
    fixList((ll:List(PAT)):PAT+->reduce("*", ll), us::List(PAT), lb, la)
  (v := isQuotient p) case Record(num:PAT, den:PAT) =>
    vr := v::Record(num:PAT, den:PAT)
    (fn := fixPredicates(vr.num, lb, la)) case "failed" => "failed"
    la := (fn::REC).res
    (fd := fixPredicates(vr.den, lb, la)) case "failed" => "failed"
    [(fn::REC).pat / (fd::REC).pat, (fd::REC).res]
  (v:= isExpt p) case Record(val:PAT, exponent:NonNegativeInteger) =>
    wr := w::Record(val:PAT, exponent: NonNegativeInteger)
    (f := fixPredicates(wr.val, lb, la)) case "failed" => "failed"
    [(f::REC).pat ** wr.exponent, (f::REC).res]
(uu := isPower p) case Record(val:PAT, exponent: PAT) =>
  uur := uu::Record(val: PAT, exponent:PAT)
  (fv := fixPredicates(uur.val, lb, la)) case "failed" => "failed"
  la := (fv::REC).res
  (fe := fixPredicates(uur.exponent, lb, la)) case "failed" =>
    "failed"
  [(fv::REC).pat ** (fe::REC).pat, (fe::REC).res]
generic? p =>
  (ua := inA(p, lb)) case "failed" => "failed"
  lp := [if (h := fixPred g) case Any then h::Any else
    return "failed" for g in predicates p]$List(Any)
  q := setPredicates(patternVariable(retract p, constant? p,
    optional? p, multiple? p), lp)
  [q, (empty?(ua::List A) => la; insertMatch(q,first(ua::List A), la))]
error "Should not happen"

——

"PMDOWN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMDOWN"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMDOWN" -> "FLAGG"

——

package PMQFCAT PatternMatchQuotientFieldCategory

—— PatternMatchQuotientFieldCategory.input ——

)set break resume
)sys rm -f PatternMatchQuotientFieldCategory.output
)spool PatternMatchQuotientFieldCategory.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PatternMatchQuotientFieldCategory
--E 1

)spool
)lisp (bye)
This package provides pattern matching functions on quotients.

See Also:
- )show PatternMatchQuotientFieldCategory

Exports:
- patternMatch

)abbrev package PMQFCAT PatternMatchQuotientFieldCategory
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 20 June 1991
++ Description:
++ This package provides pattern matching functions on quotients.

PatternMatchQuotientFieldCategory(S,R,Q):Exports == Implementation where
  S: SetCategory
  R: Join(IntegralDomain, PatternMatchable S, ConvertibleTo Pattern S)
  Q: QuotientFieldCategory R
PAT ==> Pattern S
PRQ ==> PatternMatchResult(S, Q)

Exports ==> with
  patternMatch: (Q, PAT, PRQ) -> PRQ
  ++ patternMatch(a/b, pat, res) matches the pattern pat to the
  ++ quotient a/b; res contains the variables of pat which
  ++ are already matched and their matches.

Implementation ==> add
  import PatternMatchPushDown(S, R, Q)

  patternMatch(x, p, l) ==
  generic? p => addMatch(p, x, l)
  (r := retractIfCan x)@Union(R, "failed") case R =>
    patternMatch(r::R, p, l)
  (u := isQuotient p) case Record(num:PAT, den:PAT) =>
    ur := u::Record(num:PAT, den:PAT)
    failed?(l := patternMatch(numer x, ur.num, l)) => l
    patternMatch(denom x, ur.den, l)
    failed()

——

— PMQFCAT.dotabb —

"PMQFCAT" [color="FF4488",href="bookvol10.4.pdf#nameddest=PMQFCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PMQFCAT" -> "PFECAT"

——

package PATRES2 PatternMatchResultFunctions2

— PatternMatchResultFunctions2.input —

)set break resume
)sys rm -f PatternMatchResultFunctions2.output
)spool PatternMatchResultFunctions2.output
)set message test on
)set message auto off
)clear all
PatternMatchResultFunctions2 (PATRES2)

Exports:
map

--- package PATRES2 PatternMatchResultFunctions2 ---

)abbrev package PATRES2 PatternMatchResultFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 14 Dec 1989
++ Description:
++ Lifts maps to pattern matching results.

PatternMatchResultFunctions2(R, A, B): Exports == Implementation where
  R: SetCategory
  A: SetCategory
  B: SetCategory

Exports ==> with
  map: (A -> B, PatternMatchResult(R, A)) -> PatternMatchResult(R, B)
  ++ map(f, [(v1,a1),..., (vn,an)]) returns the matching result
  ++ [(v1,f(a1)),..., (vn,f(an))].

Implementation ==> add
  map(f, r) ==
    failed? r => failed()
    construct [[rec.key, f(rec.entry)] for rec in destruct r]

— PATRES2.dotabb —

"PATRES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATRES2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"PATRES2" -> "BASTYPE"
"PATRES2" -> "KOERCE"

— package PMSYM PatternMatchSymbol —

— PatternMatchSymbol.input —

)set break resume
)sys rm -f PatternMatchSymbol.output
)spool PatternMatchSymbol.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchSymbol
PatternMatchSymbol (PMSYM)

Exports:
  patternMatch

— package PMSYM PatternMatchSymbol —

)abbrev package PMSYM PatternMatchSymbol
++ Author: Manuel Bronstein
++ Date Created: 9 Jan 1990
++ Date Last Updated: 20 June 1991
++ Description:
++ This package provides pattern matching functions on symbols.

PatternMatchSymbol(S:SetCategory): with
  patternMatch: (Symbol, Pattern S, PatternMatchResult(S, Symbol)) ->
  PatternMatchResult(S, Symbol)
  ++ patternMatch(expr, pat, res) matches the pattern pat to the
  ++ expression expr; res contains the variables of pat which
  ++ are already matched and their matches (necessary for recursion).
  == add
  import TopLevelPatternMatchControl

  patternMatch(s, p, l) ==
    generic? p => addMatch(p, s, l)
    constant? p =>
      ((u := retractIfCan(p)@Union(Symbol, "failed")) case Symbol)
      and (u::Symbol) = s => l
      failed()
    failed()

— **PMSYM.dotabb** —

"PMSYM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMSYM"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PMSYM" -> "ALIST"

— **package PMTOOLS PatternMatchTools** —

)set break resume
)sys rm -f PatternMatchTools.output
)spool PatternMatchTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchTools
--E 1
PatternMatchTools examples

This package provides tools for the pattern matcher.

See Also:
- )show PatternMatchTools

PatternMatchTools (PMTOOLS)

Exports:
- patternMatch
- patternMatchTimes

package PMTOOLS PatternMatchTools —

)abbrev package PMTOOLS PatternMatchTools
++ Author: Manuel Bronstein
++ Date Created: 13 Mar 1990
++ Date Last Updated: 4 February 1992
++ Description:
++ This package provides tools for the pattern matcher.
PatternMatchTools(S, R, P): Exports == Implementation where
  S: SetCategory
  R: Join(Ring, OrderedSet)
  P: Join(Ring, ConvertibleTo Pattern S, RetractableTo R)

  PAT ==> Pattern S
  PRS ==> PatternMatchResult(S, P)
  REC ==> Record(res:PRS, s:List P)
  RC ==> Record(pat:List PAT, s:List P)

Exports ==> with

  patternMatch: (List P, List PAT, List P -> P, PRS, (P, PAT, PRS) -> PRS) -> PRS
    ++ patternMatch(lsubj, lpat, op, res, match) matches the list
    ++ of patterns lpat to the list of subjects lsubj, allowing for
    ++ commutativity; op is the operator such that op(lpat) should
    ++ match op(lsubj) at the end, r contains the previous matches,
    ++ and match is a pattern-matching function on P.

  patternMatchTimes: (List P, List PAT, PRS, (P, PAT, PRS) -> PRS) -> PRS
    ++ patternMatchTimes(lsubj, lpat, res, match) matches the
    ++ product of patterns \spad{reduce(*,lpat)}
    ++ to the product of subjects \spad{reduce(*,lsubj)};
    ++ r contains the previous matches
    ++ and match is a pattern-matching function on P.

Implementation ==> add

  import PatternFunctions1(S, P)

  preprocessList: (PAT, List P, PRS) -> Union(List P, "failed")
  selBestGen : List PAT -> List PAT
  negConstant : List P -> Union(P, "failed")
  findMatch : (PAT, List P, PRS, P, (P, PAT, PRS) -> PRS) -> REC
  tryToMatch : (List PAT, REC, P, (P, PAT, PRS) -> PRS) ->
    Union(REC, "failed")

  filterMatchedPatterns: (List PAT, List P, PRS) -> Union(RC, "failed")

  mn1 := convert(-1::P)@Pattern(S)

  negConstant 1 ==
    for x in 1 repeat
      ((r := retractIfCan(x)@Union(R, "failed")) case R) and
        (r::R < 0) => return x
    "failed"

  -- tries to match the list of patterns lp to the list of subjects rc.s
  -- with rc.res being the list of existing matches.
  -- updates rc with the new result and subjects still to match
  tryToMatch(lp, rc, ident, pmatch) ==
    rec:REC := [l := rc.res, ls := rc.s]
for p in lp repeat
rec := findMatch(p, ls, l, ident, pmatch)
failed?(l := rec.res) => return "failed"
ls := rec.s

-- handles -1 in the pattern list.
patternMatchTimes(ls, lp, l, pmatch) ==
member?(mn1, lp) =>
  (u := negConstant ls) case "failed" => failed()
  if (u::P ^= -1::P) then ls := concat(-u::P, ls)
  patternMatch(remove(u::P,ls), remove(mn1,lp),
  (l1:List(P))::P +/- /*/l1, l, pmatch)
patternMatch(ls, lp, (l1:List(P))::P +/- /*/l1, l, pmatch)

-- finds a match for p in ls, try not to match to a "bad" value
findMatch(p, ls, l, ident, pmatch) ==
bad:List(P) :=
generic? p => setIntersection(badValues p, ls)
empty()
l1:PRS := failed()
for x in setDifference(ls, bad)
  while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
failed? l1 =>
  for x in bad
    while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
  failed? l1 => [addMatchRestricted(p, ident, l, ident), ls]
  [l1, remove(t, ls)]

-- filters out pattern if it's generic and already matched.
preprocessList(pattern, ls, l) ==
generic? pattern =>
  (u := getMatch(pattern, l)) case P =>
    member?(u::P, ls) => [u::P]
  "failed"
  empty()

-- take out already matched generic patterns
filterMatchedPatterns(lp, ls, l) ==
for p in lp repeat
  (rc := preprocessList(p, ls, l)) case "failed" => return "failed"
  if not empty?(rc::List(P)) then
    lp := remove(p, lp)
    ls := remove(first(rc::List(P)), ls)
  [lp, ls]

-- select a generic pattern with no predicate if possible
selBestGen l ==
ans := empty()$List(PAT)
for p in l | generic? p repeat
    ans := [p]
    not hasPredicate? p => return ans
ans

-- matches unordered lists ls and lp
patternMatch(ls, lp, op, 1, pmatch) ==
ident := op empty()
(rc := filterMatchedPatterns(lp, ls, 1)) case "failed" => return failed()
lp := (rc::RC).pat
ls := (rc::RC).s
empty? lp => l
#(lp := select(optional?, lp)) > 1 =>
    error "More than one optional pattern in sum/product"
(#ls + #lp) < #lp => failed()
if (not empty? lp) and (#ls + 1 = #lp) then
    lp := remove(first lp, lp)
    failed?(l := addMatch(first lp, ident, 1)) => return l
#(lp := select(multiple?, lp)) > 1 =>
    error "More than one expandable pattern in sum/product"
#ls > #lp and empty? lpm and empty?(lpm := selBestGen lp) =>
    failed()
if not empty? lp then lpm := remove(first lp, lp)
-- this is the order in which we try to match predicates
-- 11 = constant patterns (i.e. 'x, or sin('x))
11 := select(constant?, lp)
-- 12 = patterns with a predicate attached to them
12 := select((p1:PAT):Boolean+->hasPredicate? p1 and not constant? p1,lp)
-- 13 = non-generic patterns without predicates
13 := sort_!((z1:PAT,z2:PAT):Boolean+->depth(z1) > depth(z2),
    select((p2:PAT):Boolean+->not(hasPredicate? p2
        or generic? p2 or constant? p2),lp))
-- 14 = generic patterns with predicates
14 := select((p1:PAT):Boolean +-> generic? p1 and
    not(hasPredicate? p1 or constant? p1), lp)
rec:=REC := [1, ls]
(u := tryToMatch(11, rec, ident, pmatch)) case "failed" =>
    failed()
(u := tryToMatch(12, u::REC, ident, pmatch)) case "failed" =>
    failed()
(u := tryToMatch(13, u::REC, ident, pmatch)) case "failed" =>
    failed()
rec := u::REC
(rc := filterMatchedPatterns(14,rec.s,rec.res)) case "failed" => failed()
rec := [rec.res, (rc::RC).s]
(u := tryToMatch((rc::RC).pat,rec,ident,pmatch)) case "failed" => failed()
rec := u::REC
l := rec.res
ls := rec.s
empty? lpm =>
  empty? ls => 1
  failed()
  addMatch(first lpm, op ls, 1)

— PMTOOLS.dotabb —

"PMTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMTOOLS"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMTOOLS" -> "FLAGG"

— package PERMAN Permanent —

— Permanent.input —

)set break resume
)spool Permanent.output
)set message test on
)set message auto off
)clear all
--S 1 of 3
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
    r.i.i := 0
  r
--R
--R Type: Void
--E 1

--S 2 of 3
permanent(kn(5) :: SQMATRIX(5,INT))
--R
--R Compiling function kn with type PositiveInteger -> Matrix(Integer)
--R
--R (2) 44
--R Type: PositiveInteger
--E 2

--S 3 of 3
Permanent examples

The package Permanent provides the function permanent for square matrices. The permanent of a square matrix can be computed in the same way as the determinant by expansion of minors except that for the permanent the sign for each element is 1, rather than being 1 if the row plus column indices is positive and -1 otherwise. This function is much more difficult to compute efficiently than the determinant.

An example of the use of permanent is the calculation of the n-th derangement number, defined to be the number of different possibilities for n couples to dance but never with their own spouse.

Consider an n by n matrix with entries 0 on the diagonal and 1 elsewhere. Think of the rows as one-half of each couple (for example, the males) and the columns the other half. The permanent of such a matrix gives the desired derangement number.

```
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
      r.i.i := 0
  r
```

Here are some derangement numbers, which you see grow quite fast.

```
permanent(kn(5) :: SQMATRIX(5,INT))
```

```
[permanent(kn(n) :: SQMATRIX(n,INT)) for n in 1..13]
```
See Also:
- show Permanent

---

Permanent (PERMAN)

Exports:
- permanent

---

)abbrev package PERMAN Permanent
++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ References:
  ++ Henryk Minc: Evaluation of Permanents,
  ++ Nijenhuis and Wilf: Combinatorial Algorithms, Academic
  ++ S.G.Williamson, Combinatorics for Computer Science,
++ Description:
  ++ Permanent implements the functions permanent, the
  ++ permanent for square matrices.

Permanent(n : PositiveInteger, R : Ring with commutative("*")):
public == private where
  I ==> Integer
  L ==> List
  V ==> Vector
  SM ==> SquareMatrix(n,R)
VECTPKG1 ==> VectorPackage1(I)

NNI ==> NonNegativeInteger

PI ==> PositiveInteger

GRAY ==> GrayCode

public ==> with

permanent: SM -> R
++ permanent(x) computes the permanent of a square matrix x.
++ The permanent is equivalent to
++ the \spadfun{determinant} except that coefficients have
++ no change of sign. This function
++ is much more difficult to compute than the
++ determinant. The formula used is by H.J. Ryser,
++ improved by [Nijenhuis and Wilf, Ch. 19].
++ Note that permanent(x) choose one of three algorithms, depending
++ on the underlying ring R and on n, the number of rows (and
++ columns) of x:
++ if 2 has an inverse in R we can use the algorithm of
++ [Nijenhuis and Wilf, ch.19,p.158]; if 2 has no inverse,
++ some modifications are necessary:
++ if n > 6 and R is an integral domain with characteristic
++ different from 2 (the algorithm works if and only 2 is not a
++ zero-divisor of R and characteristic(R) ̸= 2,
++ but how to check that for any given R ?),
++ the local function permanent2 is called;
++ else, the local function permanent3 is called
++ (works for all commutative rings R).

private ==> add

-- local functions:

permanent2: SM -> R

permanent3: SM -> R

x : SM
a,b : R
i,j,k,l : I

permanent3(x) ==
++ This algorithm is based upon the principle of inclusion-
++ exclusion. A Gray-code is used to generate the subsets of
++ 1,...,n. This reduces the number of additions needed in
++ every step.

sgn : R := 1
k : R
a := 0$R
vv : V V I := firstSubsetGray(n)$GRAY
permanent(x) ==
  -- If 2 has an inverse in R, we can spare half of the calcu-
  -- lation needed in "permanent3": This is the algorithm of
  -- [Nijenhuis and Wilf, ch.19, p.158]
  n = 1 => x(1,1)
  two : R := (2: I) :: R
  half : Union(R,"failed") := recip(two)
  if (half case "failed") then
    if n < 7 then return permanent3(x)
    else return permanent2(x)
  sgn : R := 1
  a := 0$R
  w : V R := new(n,0$R)
  rowi : V R := new(n,0$R)
  for i in 1..n repeat
    rowi := row(x,i) :: V R
    b := 0$R
    for j in 1..n repeat
      b := b + rowi.j
    w.i := rowi(n) - (half*b)$R
  vv : V V I := firstSubsetGray((n-1): PI)$GRAY
  -- For the meaning of the elements of vv, see GRAY.
  n :: I
  b := 1
  for i in 1..n repeat
    b := b * w.i
  a := a+b
  j := 1 -- Will be the number of the element changed in subset
  while j ^= n repeat -- we sum over all subsets of (1,...,n-1)
    sgn := -sgn
    b := sgn
    if vv.1.j = 1 then k := -1
    else k := 1 -- was that element deleted(k=-1) or added(k=1)?
    for i in 1..(n::I) repeat
      w.i := w.i +$R k *$R x(i,j)
      b := b *$R w.i
      a := a +$R b
    vv := nextSubsetGray(vv,n)$GRAY
    j := vv.2.1
  if odd?(n) then a := -a
  a
b := sgn
if vv.1.j = 1 then k := -1
else k := 1 -- was that element deleted (k=-1) or added (k=1)?
for i in 1..n repeat
  w.i := w.i +$R k *$R x(i,j)
  b := b *$R w.i
  a := a +$R b
  vv := nextSubsetGray(vv,(n-1) : PI)$GRAY
  j := vv.2.1
if not odd?(n) then a := -a
two * a

permanent2(x) ==
c : R := 0
sgn : R := 1
if (not (R has IntegralDomain))
  -- or (characteristic()$R = (2:NNI))
  -- compiler refuses to compile the line above !!
  or (sgn + sgn = c)
then return permanent3(x)
-- This is a slight modification of permanent which is
-- necessary if 2 is not zero or a zero-divisor in R, but has
-- no inverse in R.
n = 1 => x(1,1)
two : R := (2:I) :: R
a := 0$R
w : V R := new(n,0$R)
-- w.i will be at first x.i and later lambda.i in
-- [Nijenhuis and Wilf, p.158, (24a) resp.(26)].
rowi : V R := new(n,0$R)
for i in 1..n repeat
  rowi := row(x,i) :: V R
  b := 0$R
  for j in 1..n repeat
    b := b + rowi.j
  w.i := (two*(rowi(n)))$R - b
vv : V V I := firstSubsetGray((n-1) : PI)$GRAY
n :: I
b := 1
for i in 1..n repeat
  b := b *$R w.i
  a := a +$R b
j := 1 -- Will be the number of the element changed in subset
while j ^= n repeat -- we sum over all subsets of (1,...,n-1)
  sgn := -sgn
  b := sgn
  if vv.1.j = 1 then k := -1
  else k := 1 -- was that element deleted (k=-1) or added (k=1)?
c := k * two
for i in 1..n repeat
\[
\begin{align*}
  w.i & := w.i + R c \times R x(i,j) \\
  b & := b \times R w.i \\
  a & := a + R b \\
  v v & := \text{nextSubsetGray}(v v, (n-1) : \text{PL}) \times \text{GRAY} \\
  j & := v v.2.1 \\
  \text{if not odd?}(n) & \text{then } a := -a \\
  b & := \text{two} \times ((n-1) : \text{NNI}) \\
  (a \text{exquo } b) & := R
\end{align*}
\]

——

— PERMAN.dotabb —

"PERMAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PERMAN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"PERMAN" -> "IVECTOR"

——

package PGE PermutationGroupExamples

— PermutationGroupExamples.input —

)set break resume
)sys rm -f PermutationGroupExamples.output
)spool PermutationGroupExamples.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PermutationGroupExamples
--E 1

)spool
)lisp (bye)

——

— PermutationGroupExamples.help —

====================================================================
PermutationGroupExamples examples
PermutationGroupExamples provides permutation groups for some classes of groups: symmetric, alternating, dihedral, cyclic, direct products of cyclic, which are in fact the finite abelian groups of symmetric groups called Young subgroups. Furthermore, Rubik's group as permutation group of 48 integers and a list of sporadic simple groups derived from the atlas of finite groups.

See Also:
o )show PermutationGroupExamples

Exports:
  abelianGroup alternatingGroup cyclicGroup dihedralGroup janko2
  mathieu11 mathieu12 mathieu22 mathieu23 mathieu24
  rubiksGroup symmetricGroup youngGroup

— package PGE PermutationGroupExamples —

)abbrev package PGE PermutationGroupExamples
++ Authors: M. Weller, G. Schneider, J. Grabmeier
++ Date Created: 20 February 1990
++ Date Last Updated: 09 June 1990
++ References:
++  J. Conway, R. Curtis, S. Norton, R. Parker, R. Wilson:
++ Description:
++ PermutationGroupExamples provides permutation groups for
++ some classes of groups: symmetric, alternating, dihedral, cyclic,
++ direct products of cyclic, which are in fact the finite abelian groups
CHAPTER 17.  CHAPTER P

++ of symmetric groups called Young subgroups.
++ Furthermore, Rubik's group as permutation group of 48 integers and a list
++ of sporadic simple groups derived from the atlas of finite groups.

PermutationGroupExamples():public == private where

L ==> List
I ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
PERM ==> Permutation
PERMGRP ==> PermutationGroup

public ==> with

symmetricGroup: PI -> PERMGRP I
++ symmetricGroup(n) constructs the symmetric group Sn
++ acting on the integers 1,...,n, generators are the
++ n-cycle (1,...,n) and the 2-cycle (1,2).
symmetricGroup: L I -> PERMGRP I
++ symmetricGroup(li) constructs the symmetric group acting on
++ the integers in the list li, generators are the
++ cycle given by li and the 2-cycle (li.1,li.2).
++ Note that duplicates in the list will be removed.
alternatingGroup: PI -> PERMGRP I
++ alternatingGroup(n) constructs the alternating group An
++ acting on the integers 1,...,n, generators are in general the
++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
++ if n is odd and the product of the 2-cycle (1,2) with
++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
++ if n is even.
alternatingGroup: L I -> PERMGRP I
++ alternatingGroup(li) constructs the alternating group acting
++ on the integers in the list li, generators are in general the
++ n-2-cycle (li.3,...,li.n) and the 3-cycle
++ (li.1,li.2,li.3), if n is odd and
++ product of the 2-cycle (li.1,li.2) with
++ n-2-cycle (li.3,...,li.n) and the 3-cycle
++ (li.1,li.2,li.3), if n is even.
++ Note that duplicates in the list will be removed.
abelianGroup: L PI -> PERMGRP I
++ abelianGroup([n1,...,nk]) constructs the abelian group that
++ is the direct product of cyclic groups with order ni.
cyclicGroup: PI -> PERMGRP I
++ cyclicGroup(n) constructs the cyclic group of order n acting
++ on the integers 1,...,n.
cyclicGroup: L I -> PERMGRP I
++ cyclicGroup([i1,...,ik]) constructs the cyclic group of
++ order k acting on the integers i1,...,ik.
++ Note that duplicates in the list will be removed.
dihedralGroup: PI -> PERMGRP I
++ dihedralGroup(n) constructs the dihedral group of order 2n
++ acting on integers 1,...,N.
dihedralGroup: L I -> PERMGRP I
++ dihedralGroup([i1,...,ik]) constructs the dihedral group of
++ order 2k acting on the integers out of i1,...,ik.
++ Note that duplicates in the list will be removed.
mathieu11: L I -> PERMGRP I
++ mathieu11(li) constructs the mathieu group acting on the 11
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ error, if li has less or more than 11 different entries.
mathieu11: () -> PERMGRP I
++ mathieu11 constructs the mathieu group acting on the
++ integers 1,...,11.
mathieu12: L I -> PERMGRP I
++ mathieu12(li) constructs the mathieu group acting on the 12
++ integers given in the list li.
++ Note that duplicates in the list will be removed
++ Error: if li has less or more than 12 different entries.
mathieu12: () -> PERMGRP I
++ mathieu12 constructs the mathieu group acting on the
++ integers 1,...,12.
mathieu22: L I -> PERMGRP I
++ mathieu22(li) constructs the mathieu group acting on the 22
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 22 different entries.
mathieu22: () -> PERMGRP I
++ mathieu22 constructs the mathieu group acting on the
++ integers 1,...,22.
mathieu23: L I -> PERMGRP I
++ mathieu23(li) constructs the mathieu group acting on the 23
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 23 different entries.
mathieu23: () -> PERMGRP I
++ mathieu23 constructs the mathieu group acting on the
++ integers 1,...,23.
mathieu24: L I -> PERMGRP I
++ mathieu24(li) constructs the mathieu group acting on the 24
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 24 different entries.
mathieu24: () -> PERMGRP I
++ mathieu24 constructs the mathieu group acting on the
++ integers 1,...,24.
janko2: L I -> PERMGRP I
++ janko2(li) constructs the janko group acting on the 100
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 100 different entries

janko2: () -> PERMGRP I
++ janko2 constructs the janko group acting on the
++ integers 1, ..., 100.

rubiksGroup: () -> PERMGRP I
++ rubiksGroup constructs the permutation group representing
++ Rubic’s Cube acting on integers 10*i+j for
++ 1 <= i <= 6, 1 <= j <= 8.
++ The faces of Rubik’s Cube are labelled in the obvious way
++ Front, Right, Up, Down, Left and Back and numbered from 1 to 6
++ in this given ordering, the pieces on each face
++ (except the unmoveable center piece) are clockwise numbered
++ from 1 to 8 starting with the piece in the upper left
++ corner. The moves of the cube are represented as permutations
++ on these pieces, represented as a two digit
++ integer ij where i is the number of the face (1 to 6)
++ and j is the number of the piece on this face.
++ The remaining ambiguities are resolved by looking
++ at the 6 generators, which represent a 90 degree turn of the
++ faces, or from the following pictorial description.
++ Permutation group representing Rubic’s Cube acting on integers
++ 10*i+j for 1 <= i <= 6, 1 <= j <= 8.
++
++ \begin{verbatim}
++ Rubik’s Cube: +-----+ +-- B where: marks Side #: 
++ / U /|/
++ / / | F(ront) <-> 1
++ L --> +-----+ R| R(ight) <-> 2
++ | | + U(p) <-> 3
++ | F | / D(own) <-> 4
++ | | / L(eft) <-> 5
++ +-----+ B(ack) <-> 6
++ |
++ D
++
++ The Cube’s surface:
++ The pieces on each side
++ (except the unmoveable center
++ piece) are clockwise numbered
++ from 1 to 8 starting with the
++ piece in the upper left
++ corner (see figure on the
++ left). The moves of the cube
++ are represented as
++ permutations on these pieces.
++ Each of the pieces is
++ represented as a two digit
++ integer ij where i is the
++ |765| # of the side (1 to 6 for
++ +---+ F to B (see table above)
++ |567| and j is the # of the piece.
++ |4B8|
++ |321|
++ +---+
++ \end{verbatim}
youngGroup: L I -> PERMGRP I
++ youngGroup([n1,...,nk]) constructs the direct product of the
++ symmetric groups Sn1,...,Snk.
youngGroup: Partition -> PERMGRP I
++ youngGroup(lambda) constructs the direct product of the symmetric
++ groups given by the parts of the partition lambda.

private ==> add

-- import the permutation and permutation group domains:

import PERM I
import PERMGRP I

-- import the needed map function:

import ListFunctions2(L L L I,PERM I)

-- the internal functions:

\[
\text{llli2gp(l:L L L I)}:\text{PERMGRP I} == \\
\text{---+ Converts an list of permutations each represented by a list} \\
\text{---+ of cycles ( each of them represented as a list of Integers )} \\
\text{---+ to the permutation group generated by these permutations.} \\
\text{(map(cycles,l))::PERMGRP I}
\]

\[
\text{li1n(n:I):L I ==} \\
\text{---+ constructs the list of integers from 1 to n} \\
[\text{i for i in 1..n}]
\]

-- definition of the exported functions:

youngGroup(l:L I):PERMGRP I ==
\[
\text{gens:= nil()$(L L L I)} \\
\text{element:= 1} \\
\text{for n in l | n > 1 repeat} \\
\text{gens:=cons(list [i for i in element..(element+n-1)], gens)} \\
\text{if n >= 3 then gens := cons([[element,element+1]],gens)} \\
\text{element:=element+n} \\
\text{llli2gp} \\
\text{#gens = 0 => [[[1]]]} \\
\text{gens}
\]
youngGroup(lambda : Partition):PERMGRP I ==
youngGroup(convert(lambda)$Partition)
rubiksGroup():PERMGRP I ==
  -- each generator represents a 90 degree turn of the appropriate
  -- side.
f:L L I:=
  [[11,13,15,17], [12,14,16,18], [51,31,21,41], [53,33,23,43], [52,32,22,42]]
r:L L I:=
  [[21,23,25,27], [22,24,26,28], [13,37,67,43], [15,31,61,45], [14,38,68,44]]
u:L L I:=
  [[31,33,35,37], [32,34,36,38], [13,51,63,25], [11,57,61,23], [12,58,62,24]]
d:L L I:=
  [[41,43,45,47], [42,44,46,48], [17,21,67,55], [15,27,65,53], [16,28,66,54]]
l:L L I:=
  [[51,53,55,57], [52,54,56,58], [11,41,65,35], [17,47,63,33], [18,48,64,34]]
b:L L I:=
  [[61,63,65,67], [62,64,66,68], [45,25,35,55], [47,27,37,57], [46,26,36,56]]
llli2gp [f,r,u,d,l,b]

mathieu11(l:L I):PERMGRP I ==
  -- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 11 => error "Exactly 11 integers for mathieu11 needed !"
  a:L L I:=
  [[l.1,l.10],[l.2,l.8],[l.3,l.11],[l.5,l.7]]
  lllli2gp [a,[[1.1,1.4,1.7,1.6],[1.2,1.11,1.10,1.9]]]

mathieu11():PERMGRP I == mathieu11 li1n 11

mathieu12(l:L I):PERMGRP I ==
  -- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 12 => error "Exactly 12 integers for mathieu12 needed !"
  a:L L I:=
  [[1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,1.10,1.11]]
  lllli2gp [a,[[1.1,1.6,1.5,1.8,1.3,1.7,1.4,1.2,1.9,1.10],[1.11,1.12]]]

mathieu12():PERMGRP I == mathieu12 li1n 12

mathieu22(l:L I):PERMGRP I ==
  -- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 22 => error "Exactly 22 integers for mathieu22 needed !"
  a:L L I:=
  [[1.1,1.2,1.4,1.8,1.16,1.9,1.18,1.13,1.3,1.6,1.12],
   [1.5,1.10,1.17,1.11,1.22,1.21,1.19,1.15,1.7,1.14]]
  b:L L I:=
  [[1.1,1.2,1.6,1.18],[1.3,1.15],[1.5,1.8,1.21,1.13],[1.7,1.9,1.20,1.12],[1.10,1.16],[1.11,1.19,1.14,1.22]]
  lllli2gp [a,b]

mathieu22():PERMGRP I == mathieu22 li1n 22

mathieu23(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
l := removeDuplicates l
#1 ^= 23 => error "Exactly 23 integers for mathieu23 needed !"
a := [[l.1, l.2, l.3, l.4, l.5, l.6, l.7, l.8, l.9, l.10, l.11, l.12, l.13, l.14, l.15, l.16, l.17, l.18, l.19, l.20, l.21, l.22, l.23], [l.1, l.2, l.3, l.4, l.5, l.6, l.7, l.8, l.9, l.10, l.11, l.12, l.13, l.14, l.15, l.16, l.17, l.18, l.19, l.20, l.21, l.22, l.23]
llli2gp [a, b]

mathieu23(): PERMGRP I == mathieu23 li1n 23

mathieu24(l: L I): PERMGRP I ==
-- permutations derived from the ATLAS
l := removeDuplicates l
#1 ^= 24 => error "Exactly 24 integers for mathieu24 needed !"
a := [[l.1, l.16, l.10, l.22, l.24], [l.2, l.12, l.18, l.21, l.7], [l.4, l.5, l.8, l.6, l.17], [l.9, l.11, l.13, l.19, l.15], [l.1, l.2, l.3, l.4, l.5, l.6, l.7, l.8, l.9, l.10, l.11, l.12, l.13, l.14, l.15, l.16, l.17, l.18, l.19, l.20, l.21, l.22, l.23, l.24]]
llli2gp [a, b]

mathieu24(): PERMGRP I == mathieu24 li1n 24

janko2(l: L I): PERMGRP I ==
-- permutations derived from the ATLAS
l := removeDuplicates l
#1 ^= 100 => error "Exactly 100 integers for janko2 needed !"
a := [[l.2, l.3, l.4, l.5, l.6, l.7, l.8], [l.9, l.10, l.11, l.12, l.13, l.14, l.15], [l.16, l.17, l.18, l.19, l.20, l.21, l.22], [l.23, l.24, l.25, l.26, l.27, l.28, l.29], [l.30, l.31, l.32, l.33, l.34, l.35, l.36], [l.37, l.38, l.39, l.40, l.41, l.42, l.43], [l.44, l.45, l.46, l.47, l.48, l.49, l.50], [l.51, l.52, l.53, l.54, l.55, l.56, l.57], [l.58, l.59, l.60, l.61, l.62, l.63, l.64], [l.65, l.66, l.67, l.68, l.69, l.70, l.71], [l.72, l.73, l.74, l.75, l.76, l.77, l.78], [l.79, l.80, l.81, l.82, l.83, l.84, l.85], [l.86, l.87, l.88, l.89, l.90, l.91, l.92], [l.93, l.94, l.95, l.96, l.97, l.98, l.99]]
[1.10,1.78,1.88,1.29,1.12] ]
lli2gp [a,b]

janko2():PERMGRP I == janko2 li1n 100

abelianGroup(l:L PI):PERMGRP I ==
  gens:= nil()$(L L L I)
  element:I:= 1
  for n in l | n > 1 repeat
    gens:=cons( list [i for i in element..(element+n-1) ], gens )
    element:=element+n
  llli2gp
  #gens = 0 => [[[1]]]
  gens

alternatingGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 =>
    error "Cannot construct alternating group on empty set"
  #l < 3 => llli2gp [[[1]]]
  #l = 3 => llli2gp [[[1,1,1,2,1,3]]]
  tmp:= [l.i for i in 3..(#l-1)]
  gens:L L L I:=[[tmp],[[1,1,1,2,1,3]]]
  odd?(#l) => llli2gp gens
  gens.1 := cons([1,1,1,2],gens.1)
  llli2gp gens

alternatingGroup(n:PI):PERMGRP I == alternatingGroup li1n n

symmetricGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 => error "Cannot construct symmetric group on empty set !"
  #l < 3 => llli2gp [[1]]
  llli2gp [[1],[[1,1,1,2]]]

symmetricGroup(n:PI):PERMGRP I == symmetricGroup li1n n

cyclicGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 => error "Cannot construct cyclic group on empty set"
  llli2gp [[1]]

cyclicGroup(n:PI):PERMGRP I == cyclicGroup li1n n

dihedralGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l < 3 => error "in dihedralGroup: Minimum of 3 elements needed !"
  tmp := [[l.i, l.((#l-i+1) ) for i in 1..(#l quo 2)]]
  llli2gp [ [ 1 ], tmp ]
dihedralGroup(n:PI):PERMGRP I ==
n = 1 => symmetricGroup (2::PI)
n = 2 => llii2gp [[1,2]],[[3,4]]
n = n => dihedralGroup liiin n

package PICOERCE PiCoercions

---

PiCoercions.examples

Provides a coercion from the symbolic fractions in %pi with integer coefficients to any Expression type.
PiCoercions (PICOERCE)

Exports:
coerce

— package PICOERCE PiCoercions —

)abbrev package PICOERCE PiCoercions
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 21 Feb 1990
++ Description:
++ Provides a coercion from the symbolic fractions in %pi with
++ integer coefficients to any Expression type.

PiCoercions(R:Join(OrderedSet, IntegralDomain)): with
  coerce: Pi -> Expression R  
  ++ coerce(f) returns f as an Expression(R).
== add
p2e: SparseUnivariatePolynomial Integer -> Expression R

coerce(x:Pi):Expression(R) ==
  f := convert(x)$Fraction(SparseUnivariatePolynomial Integer)
  p2e(numer f) / p2e(denom f)

p2e p ==
  map((x1:Integer):Expression(R) +-> x1::Expression(R), p)_
\$\text{SparseUnivariatePolynomialFunctions2(Integer, Expression R)}\_
(\text{pi()}\$\text{Expression(R)})

---

--- PICOERCE.dotabb ---

"PICOERCE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PICOERCE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PICOERCE" -> "FS"

---

package PLOT1 PlotFunctions1

--- PlotFunctions1.input ---

)set break resume
)sys rm -f PlotFunctions1.output
)spool PlotFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PlotFunctions1
--E 1

)spool
)lisp (bye)

---

--- PlotFunctions1.help ---

====================================================================
PlotFunctions1 examples
====================================================================

PlotFunctions1 provides facilities for plotting curves
where functions SF \rightarrow SF are specified by giving an expression

See Also:
CHAPTER 17. CHAPTER P

o )show PlotFunctions1

---

PlotFunctions1 (PLOT1)

Exports:
plotPolar plot

— package PLOT1 PlotFunctions1 —

)abbrev package PLOT1 PlotFunctions1
++ Authors: R.T.M. Bronstein, C.J. Williamson
++ Date Created: Jan 1989
++ Date Last Updated: 4 Mar 1990
++ Description:
++ PlotFunctions1 provides facilities for plotting curves
++ where functions SF -> SF are specified by giving an expression

PlotFunctions1(S:ConvertibleTo InputForm): with
    plot : (S, Symbol, Segment DoubleFloat) -> Plot
    ++ plot(fcn,x,seg) plots the graph of \(y = f(x)\) on a interval
    plot : (S, S, Symbol, Segment DoubleFloat) -> Plot
    ++ plot(f,g,t,seg) plots the graph of \(x = f(t), y = g(t)\)
    ++ as \(t\) ranges over an interval.
    plotPolar : (S, Symbol, Segment DoubleFloat) -> Plot
    ++ plotPolar(f,theta,seg) plots the graph of \(r = f(\theta)\) as
    ++ \(\theta\) ranges over an interval
    plotPolar : (S, Symbol) -> Plot
    ++ plotPolar(f,theta) plots the graph of \(r = f(\theta)\) as
    ++ \(\theta\) ranges from 0 to 2 pi
== add
    import MakeFloatCompiledFunction(S)
plot(f, x, xRange) == plot(makeFloatFunction(f, x), xRange)
plotPolar(f, theta) == plotPolar(makeFloatFunction(f, theta))
plot(f1, f2, t, tRange) ==
    plot(makeFloatFunction(f1, t), makeFloatFunction(f2, t), tRange)
plotPolar(f, theta, thetaRange) ==
    plotPolar(makeFloatFunction(f, theta), thetaRange)

— PLOT1.dotabb —

"PLOT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLOT1"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"PLOT1" -> "KONVERT"

package PLOTTOOL PlotTools

— PlotTools.input —

)set break resume
)sys rm -f PlotTools.output
)spool PlotTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PlotTools
--E 1

)spool
)lisp (bye)

— PlotTools.help —

====================================================================
PlotTools examples
====================================================================
This package exports plotting tools

See Also:
  o )show PlotTools

---

PlotTools (PLOTTOOL)

Exports:
calcRanges

— package PLOTTOOL PlotTools —

)abbrev package PLOTTOOL PlotTools
++ Description:
++ This package exports plotting tools

PlotTools(): Exports == Implementation where
  L ==> List
  -- Pt ==> TwoDimensionalPoint
  SEG ==> Segment
  SF ==> DoubleFloat
  Pt ==> Point(SF)
  PLOT ==> Plot
  DROP ==> DrawOption
  S ==> String
  VIEW2D ==> TwoDimensionalViewport

Exports == with
calcRanges: L L Pt -> L SEG SF
  ++ calcRanges(l) \undocumented
Implementation ==> add
    import GraphicsDefaults
    import PLOT
    import TwoDimensionalPlotClipping
    import DrawOptionFunctions0
    import ViewportPackage
    import POINT
    import PointPackage(SF)

--%Local functions
xRange0: L Pt -> SEG SF
xRange: L L Pt -> SEG SF
yRange0: L Pt -> SEG SF
yRange: L L Pt -> SEG SF
drawToScaleRanges: (SEG SF,SEG SF) -> L SEG SF

drawToScaleRanges(xVals,yVals) ==
    xDiff := (xHi := hi xVals) - (xLo := lo xVals)
    yDiff := (yHi := hi yVals) - (yLo := lo yVals)
    pad := abs(yDiff - xDiff)/2
    yDiff > xDiff => [segment(xLo - pad,xHi + pad),yVals]
    [xVals,segment(yLo - pad,yHi + pad)]

select : (L Pt,Pt -> SF,(SF,SF) -> SF) -> SF
select(l,f,g) ==
    m := f first l
    for p in rest l repeat m := g(m,f p)
    m

xRange0(list:L Pt) == select(list,xCoord,min) .. select(list,xCoord,max)
yRange0(list:L Pt) == select(list,yCoord,min) .. select(list,yCoord,max)

select2: (L L Pt,L Pt -> SF,(SF,SF) -> SF) -> SF
select2(l,f,g) ==
    m := f first l
    for p in rest l repeat m := g(m,f p)
    m

xRange(list:L L Pt) ==
    select2(list,(u1:L(Pt)):SF +-> lo(xRange0(u1)),min) ..
    .. select2(list,(v1:L(Pt)):SF +-> hi(xRange0(v1)),max)

yRange(list:L L Pt) ==
    select2(list,(u1:L(Pt)):SF +-> lo(yRange0(u1)),min) ..
    .. select2(list,(v1:L(Pt)):SF +-> hi(yRange0(v1)),max)

--%Exported Functions
calcRanges(llp) ==
    drawToScale() => drawToScaleRanges(xRange llp, yRange llp)
package PRJALGPK ProjectiveAlgebraicSetPackage

)set break resume
)sys rm -f ProjectiveAlgebraicSetPackage.output
)spool ProjectiveAlgebraicSetPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectiveAlgebraicSetPackage
--R
--R ProjectiveAlgebraicSetPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: DirectProductCategory(#(symb),NonNegativeInteger),ProjPt: ProjectiveSpaceCategory(K)) is a package constructor
--R Abbreviation for ProjectiveAlgebraicSetPackage is PRJALGPK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRJALGPK
--R
--R-------------------------------- Operations --------------------------------
--R algebraicSet : List(PolyRing) -> List(ProjPt)
--R rationalPoints : (PolyRing,PositiveInteger) -> List(ProjPt)
--R singularPoints : PolyRing -> List(ProjPt)
--R singularPointsWithRestriction : (PolyRing,List(PolyRing)) -> List(ProjPt)
--R
--E 1

)spool
)lisp (bye)
ProjectiveAlgebraicSetPackage (PRJALGPK)

---

Exports:
algebraicSet rationalPoints singularPoints singularPointsWithRestriction

---

)abbrev package PRJALGPK ProjectiveAlgebraicSetPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt):_
Exports == Implementation where
K : Field
symb: List(Symbol)
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)

OF ==> OutputForm
PI ==> PositiveInteger
RFP ==> RootsFindingPackage
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
SPWRES ==> AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt)
SPWGRO ==> AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt)

Exports == with

  singularPointsWithRestriction: (PolyRing,List(PolyRing)) -> List(ProjPt)
  ++ return the singular points that anhilate

  singularPoints: PolyRing -> List(ProjPt)
  ++ singularPoints retourne les points singulier

  algebraicSet: List(PolyRing) -> List(ProjPt)
  ++ algebraicSet returns the algebraic set if finite (dimension 0).

  rationalPoints: (PolyRing,PI) -> List(ProjPt)
  ++ \texttt{rationalPoints(f,d)} returns all points on the curve \texttt{f}
  ++ in the extension of the ground field of degree \texttt{d}.
  ++ For \texttt{d > 1} this only works if \texttt{K} is a
  ++ \texttt{LocallyAlgebraicallyClosedField}

Implementation == add

import PPFC1
import PolyRing
import ProjPt

listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]
polyToX10 : PolyRing -> SUP(K)

--fonctions de resolution de sys. alg. de dim 0
singularPoints(crb)==
F:=crb
Fx:=differentiate(F,index(1)$OV)
Fy:=differentiate(F,index(2)$OV)
Fz:=differentiate(F,index(3)$OV)
idealT:List PolyRing:=[F,Fx,Fy,Fz]
idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT]
recOfZerosX10:= distinguishedCommonRootsOf(idealToX10,1)$RFP(K)
listOfExtDeg:List Integer:=[recOfZerosX10.extDegree]
degExt:=lcm listOfExtDeg
zero?(degExt) =>
  error("------- Infinite number of points ------")
one?(degExt) =>
  print(("You need an extension of degree")::OF)
  print(degExt::OF)
  error("----------Have a nice day----------")
listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
tempL:= affineSingularPoints(crb)$SPWRES
if tempL case "failed" then
  print(("failed with resultant")::OF)
  print("The singular points will be computed using grobner basis"::OF)
tempL := affineSingularPoints(crb)$SPWGRO
tempL case "Infinite" =>
  error("------- Infinite number of points ------")
tempL case Integer =>
  print(("You need an extension of degree")::OF)
  print(tempL ::OF)
  error("---------Have a nice day----------")
listPtsIdl2:List(ProjPt)
if tempL case List(ProjPt) then
  listPtsIdl2:= ( tempL :: List(ProjPt))
else
  error" From ProjectiveAlgebraicSetPackage: this should not happen"
listPtsIdl1 := concat( listPtsIdl , listPtsIdl2)
if pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
  listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
listPtsIdl

algebraicSet(idealT:List(PolyRing)) ==
idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT] 
recOfZerosX10:= distinguishedCommonRootsOf(idealToX10,1)$RFP(K) 
listOfExtDeg:List Integer:=[recOfZerosX10.extDegree] 
degExt:=lcm listOfExtDeg
zero?(degExt) =>
  error("------- Infinite number of points ------")
one?(degExt) =>
  print(("You need an extension of degree")::OF)
  print(degExt::OF)
  error("---------Have a nice day----------")
listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
tempL:= affineAlgSet( idealT )$SPWRES
if tempL case "failed" then
  print("failed with resultant"::OF)
  print("The finite alg. set will be computed using grobner basis"::OF)
tempL := affineAlgSet( idealT )$SPWGRO
tempL case "Infinite" =>
  error("------- Infinite number of points ------")
tempL case Integer =>
  print(("You need an extension of degree")::OF)
  print(tempL ::OF)
error("-------------Have a nice day-------------")
listPtsIdl2: List(ProjectivePoint)
if tempL case List(ProjectivePoint) then
  listPtsIdl2 := ( tempL :: List(ProjectivePoint) )
else
  error" From ProjectiveAlgebraicSetPackage: this should not happen"
listPtsIdl := concat( listPtsIdl , listPtsIdl2)
if pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
  listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
listPtsIdl

if K has FiniteFieldCategory then

  rationalPoints(crv:PolyRing,extdegree:PI):List(ProjectivePoint) ==
  --The code of this is almost the same as for algebraicSet
  --We could just construct the ideal and call algebraicSet
  --Should we do that? This might be a bit faster.
  listPtsIdl:List(ProjectivePoint):= empty()
  x:= monomial(1,1)$SUP(K)
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)$K
    q:= size()$K
    px:= x**(q**extdegree) - x
  crvX10:= polyToX10 crv
  recOfZerosX10:= distinguishedCommonRootsOf([crvX10,px],1$K)$RFP(K)
  listPtsIdl:=[projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
  --now we got all of the projective points where z = 0 and y ^= 0
  ratXY1 : List ProjectivePoint( = affineRationalPoints( crv, extdegree )$SPWGRO
  listPtsIdl:= concat(ratXY1,listPtsIdl)
  if pointInIdeal?([crv],projectivePoint([1,0,0]))$PPFC1 then
    listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
  listPtsIdl

polyToX10(pol)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  pp:= parts dd
  lp:= last pp
  "zero?(lp) => polyToX10 reductum pol
  el:= pp.1
  monomial(lc,el)$SUP(K) + polyToX10 reductum pol

singularPointsWithRestriction(F,lstPol)==
  Fx:= differentiate(F,index(1)$OV)
  Fy:= differentiate(F,index(2)$OV)
  Fz:= differentiate(F,index(3)$OV)
  idealSingulier:List(PolyRing):=concat([Fx,Fy,Fz],lstPol)
  algebraicSet(idealSingulier)
---

---

package PTFUNC2 PointFunctions2

---

PointFunctions2.input

)set break resume
)sys rm -f PointFunctions2.output
)spool PointFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointFunctions2
--E 1

)spool
)lisp (bye)

---

PointFunctions2.help

====================================================================
PointFunctions2 examples
====================================================================

This package has no description

See Also:
  o )show PointFunctions2

---
PointFunctions2 (PTFUNC2)

Exports:
map

— package PTFUNC2 PointFunctions2 —

)abbrev package PTFUNC2 PointFunctions2
++ Description:
++ This package has no description

PointFunctions2(R1:Ring,R2:Ring):Exports == Implementation where

Exports == with
  map : ((R1->R2),Point(R1)) -> Point(R2)
  ++ map(f,p) undocumented

Implementation ==> add
import Point(R1)
import Point(R2)

map(mapping,p) ==
  point([mapping p.(i::PositiveInteger) for i in minIndex(p)..maxIndex(p)])$Point(R2)

— PTFUNC2.dotabb —

"PTFUNC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PTFUNC2"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PTFUNC2" -> "PID"
"PTFUNC2" -> "OAGROUP"
package PTPACK PointPackage

--- PointPackage.input ---

)set break resume
)sys rm -f PointPackage.output
)spool PointPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointPackage
--E 1

)spool
)lisp (bye)

---

--- PointPackage.help ---

====================================================================
PointPackage examples
====================================================================

This package has no description

See Also:
o )show PointPackage

---
PointPackage (PTPACK)

Exports:
  color   hue   phiCoord  rCoord  shade
thetaCoord  xCoord  yCoord  zCoord

— package PTPACK PointPackage —

)}abbrev package PTPACK PointPackage
++ Description:
++ This package has no description

PointPackage(R:Ring):Exports == Implementation where

POINT ==> Point(R)
I ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
L ==> List
B ==> Boolean

Exports == with
  xCoord : POINT -> R
    ++ xCoord(pt) returns the first element of the point, pt,
    ++ although no assumptions are made as to the coordinate
    ++ system being used. This function is defined for the
    ++ convenience of the user dealing with a Cartesian
    ++ coordinate system.
  yCoord : POINT -> R
    ++ yCoord(pt) returns the second element of the point, pt,
    ++ although no assumptions are made as to the coordinate
    ++ system being used. This function is defined for the
    ++ convenience of the user dealing with a Cartesian
    ++ coordinate system.
  zCoord : POINT -> R
    ++ zCoord(pt) returns the third element of the point, pt,
    ++ although no assumptions are made as to the coordinate
++ system being used. This function is defined for the
++ convenience of the user dealing with a Cartesian
++ or a cylindrical coordinate system.

rCoord : POINT -> R
++ rCoord(pt) returns the first element of the point, pt,
++ although no assumptions are made as to the coordinate
++ system being used. This function is defined for the
++ convenience of the user dealing with a spherical
++ or a cylindrical coordinate system.

thetaCoord : POINT -> R
++ thetaCoord(pt) returns the second element of the point, pt,
++ although no assumptions are made as to the coordinate
++ system being used. This function is defined for the
++ convenience of the user dealing with a spherical
++ or a cylindrical coordinate system.

phiCoord : POINT -> R
++ phiCoord(pt) returns the third element of the point, pt,
++ although no assumptions are made as to the coordinate
++ system being used. This function is defined for the
++ convenience of the user dealing with a spherical
++ coordinate system.

color : POINT -> R
++ color(pt) returns the fourth element of the point, pt,
++ although no assumptions are made with regards as to
++ how the components of higher dimensional points are
++ interpreted. This function is defined for the
++ convenience of the user using specifically, color
++ to express a fourth dimension.

hue : POINT -> R
++ hue(pt) returns the third element of the two dimensional point, pt,
++ although no assumptions are made with regards as to how the
++ components of higher dimensional points are interpreted. This
++ function is defined for the convenience of the user using
++ specifically, hue to express a third dimension.

shade : POINT -> R
++ shade(pt) returns the fourth element of the two dimensional
++ point, pt, although no assumptions are made with regards as to
++ how the components of higher dimensional points are interpreted.
++ This function is defined for the convenience of the user using
++ specifically, shade to express a fourth dimension.

-- 2D and 3D extraction of data
Implementation ==> add

xCoord p == elt(p,1)
yCoord p == elt(p,2)
zCoord p == elt(p,3)
rCoord p == elt(p,1)
thetaCoord p == elt(p,2)
phiCoord p == elt(p,3)
color p ==
  #p > 3 => p.4
  p.3
hue p == elt(p,3)
  -- 4D points in 2D using extra dimensions for palette information
shade p == elt(p,4)
  -- 4D points in 2D using extra dimensions for palette information

——

— PTPACK.dotabb —

"PTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PTPACK"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"PTPACK" -> "PTCAT"

——

package PFO PointsOfFiniteOrder

—— PointsOfFiniteOrder.input —

)set break resume
)sys rm -f PointsOfFiniteOrder.output
)spool PointsOfFiniteOrder.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointsOfFiniteOrder
--E 1

)spool
)lisp (bye)

——

— PointsOfFiniteOrder.help —

====================================================================
PointsOfFiniteOrder examples
====================================================================
This package provides function for testing whether a divisor on a curve is a torsion divisor.

See Also:
o )show PointsOfFiniteOrder

---

PointsOfFiniteOrder (PFO)

Exports:
order  torsion?  torsionIfCan

--- package PFO PointsOfFiniteOrder ---

)abbrev package PFO PointsOfFiniteOrder
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 July 1998
++ Description:
++ This package provides function for testing whether a divisor on a curve is a torsion divisor.

PointsOfFiniteOrder(R0, F, UP, UPUP, R): Exports == Implementation where
  R0    : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F     : FunctionSpace R0
  UP    : UnivariatePolynomialCategory F
  UPUP  : UnivariatePolynomialCategory Fraction UP
  R     : FunctionFieldCategory(F, UP, UPUP)

PI    => PositiveInteger
N     => NonNegativeInteger
Z ==> Integer
Q ==> Fraction Integer
UPF ==> SparseUnivariatePolynomial F
UPQ ==> SparseUnivariatePolynomial Q
QF ==> Fraction UP
UPUPQ ==> SparseUnivariatePolynomial Fraction UPQ
UP2 ==> SparseUnivariatePolynomial UP
UP3 ==> SparseUnivariatePolynomial UP2
FD ==> FiniteDivisor(F, UP, UPUP, R)
K ==> Kernel F
REC ==> Record(ncurve:UP3, disc:Z, dfpoly:UPQ)
RC0 ==> Record(ncurve:UPUPQ, disc:Z)
ID ==> FractionalIdeal(UP, QF, UPUP, R)
SMP ==> SparseMultivariatePolynomial(R0,K)
ALGOP ==> "%alg"

Exports ==> with
  order : FD -> Union(N, "failed")
  ++ order(f) \undocumented
  torsion? : FD -> Boolean
  ++ torsion?(f) \undocumented
  torsionIfCan : FD -> Union(Record(order:N, function:R), "failed")
  ++ torsionIfCan(f) \ undocumented

Implementation ==> add
  import IntegerPrimesPackage(Z)
  import PointsOfFiniteOrderTools(UPQ, UPUPQ)
  import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)

  cmult: List SMP -> SMP
  raise : (UPQ, K) -> F
  raise2 : (UP2, K) -> UP
  qmod : F -> Q
  fmod : UPF -> UPQ
  rmod : UP -> UPQ
  pmod : UPUP -> UPUPQ
  kqmod : (F, K) -> UPQ
  krmod : (UP, K) -> UP2
  kpmod : (UPUP, K) -> UP3
  selectIntegers: K -> REC
  selIntegers: () -> RC0
  possibleOrder : FD -> N
  ratcurve : (FD, RC0) -> N
  algcurve : (FD, REC, K) -> N
  kbad3Num : (UP3, UPQ) -> Z
  kbadBadNum : (UP2, UPQ) -> Z
  kgetGoodPrime : (REC, UPQ, UP3, UP2, UP2) -> Record(prime:PI,poly:UPQ)
  goodRed : (REC, UPQ, UP3, UP2, UP2, PI) -> Union(UPQ, "failed")
  good? : (UPQ, UP3, UP2, UP2, PI, UPQ) -> Boolean
  klist : UP -> List K
aklist : R -> List K
alglist : FD -> List K
notIrr? : UPQ -> Boolean
rat : (UPUP, FD, PI) -> N
toQ1 : (UP2, UPQ) -> UP
toQ2 : (UP3, UPQ) -> R
Q2F : Q -> F
Q2UPUP : UPUPQ -> UPUP

q := FunctionSpaceReduce(R0, F)

torsion? d == order(d) case N
Q2F x == numer(x)::F / denom(x)::F
qmod x == bringDown(x)$q
kqmod(x,k) == bringDown(x, k)$q
fmod p == map(qmod, p)$SparseUnivariatePolynomialFunctions2(F, Q)
mod p == map(qmod, p)$MultipleMap(F, UP, UPUP, Q, UPQ, UPUPQ)
Q2UPUP p == map(Q2F, p)$MultipleMap(Q, UPQ, UPUPQ, F, UP, UPUP)
klist d == "setUnion"/[kernels c for c in coefficients d]
notIrr? d == #(factors factor(d)$RationalFactorize(UPQ)) > 1
kbadBadNum(d, m) == mix [badNum(c rem m) for c in coefficients d]
kbad3Num(h, m) == lcm [kbadBadNum(c, m) for c in coefficients h]

torsionIfCan d ==
    zero?(n := possibleOrder(d := reduce d)) => "failed"
    (g := generator reduce(n::Z * d)) case "failed" => "failed"
    [n, g::R]

UPQ2F(p:UPQ, k:K):F ==
    map(Q2F, p)$UnivariatePolynomialCategoryFunctions2(Q, UPQ, F, UP) (k::F)

UP22UP(p:UP2, k:K):UP ==
    map((p1:UPQ):F +-> UPQ2F(p1, k) + F, p)
    $UnivariatePolynomialCategoryFunctions2(UPQ,UP2,F,UP)

UP32UPUP(p:UP3, k:K):UPUP ==
    map((p1:UP2):QF +-> UP22UP(p1, k) + QF, p)
    $UnivariatePolynomialCategoryFunctions2(UPQ,UP2,F,UP)

if R0 has GcdDomain then
    cmult(l:List SMP):SMP == lcm l
else
    cmult(l:List SMP):SMP == */l

doubleDisc(f:UP3):Z ==
    d := discriminant f
    g := gcd(d, differentiate d)
    d := (d exquo g)::UP2
    zero?(e := discriminant d) => 0
    gcd [retract(c)@Z for c in coefficients e]
commonDen(p:UP):SMP ==
  l1:List F := coefficients p
  l2:List SMP := [denom c for c in l1]
  cmult l2

polyred(f:UPUP):UPUP ==
  cmult([commonDen(retract(c)@UP) for c in coefficients f])::F::UP::QF * f

aklist f ==
  (r := retractIfCan(f)@Union(QF, "failed")) case "failed" =>
  "setUnion"/[klist(retract(c)@UP) for c in coefficients lift f]
  klist(retract(r::QF)@UP)

alglist d ==
  n := numer(i := ideal d)
  select_!((k1:K):Boolean +-> has?(operator k1, ALGOP),
    setUnion(klist denom i,
      "setUnion"/[aklist qelt(n,i) for i in minIndex n..maxIndex n]))

krmod(p,k) ==
  map(z1 +-> kqmod(z1, k),
    p)$UnivariatePolynomialCategoryFunctions2(F, UP, UPQ, UP2)

rmod p ==
  map(qmod, p)$UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)

raise(p, k) ==
  (map(Q2F, p)$SparseUnivariatePolynomialFunctions2(Q, F)) (k::F)

raise2(p, k) ==
  map(z1 +-> raise(z1, k),
    p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)

algcurve(d, rc, k) ==
  mn := minIndex(n := numer(i := minimize ideal d))
  h := kqmod(lift(hh := n(mn + 1)), k)
  b2 := primitivePart
    raise2(b := krmod(retract(retract(n.mn)@QF)@UP, k), k)
  s := kqmod(resultant(primitivePart separate(raise2(krmod(
    retract(norm hh)@UP, k), k), b2).primePart, b2), k)
  pr := kgetGoodPrime(rc, s, h, b, dd := krmod(denom i, k))
  p := pr.prime
  pp := UP32UPUP(rc.ncurve, k)
  mm := pr.poly
  gf := InnerPrimeField p
  m := map((z1:Q):gf +-> retract(z1)@Z :: gf,mm)
  $SparseUnivariatePolynomialFunctions2(Q, gf)
-- one? degree m =>
(degree m = 1) =>
alpha := - coefficient(m, 0) / leadingCoefficient m
order(d, pp,
  (z1:F):gf +-> (map((q1:Q):gf +-> numer(q1) :: gf / denom(q1) :: gf,
  kqmod(z1,k))$SparseUnivariatePolynomialFunctions2(Q,gf))(alpha)
)$ReducedDivisor(F, UP, UPUP, R, gf)
-- d1 := toQ1(dd, mm)
-- rat(pp, divisor ideal([[toQ1(b, mm) / d1] :: QF :: R,
  -- inv(d1 :: QF) * toQ2(h, mm)]$ID, p)
sae := SimpleAlgebraicExtension(gf, SparseUnivariatePolynomial gf, m)
order(d, pp,
  (z1:F):sae +-> reduce(map((q1:Q):gf +-> numer(q1) :: gf / denom(q1) :: gf,
  kqmod(z1,k))$SparseUnivariatePolynomialFunctions2(Q,gf))$sae
)$ReducedDivisor(F, UP, UPUP, R, sae)

-- returns the potential order of d, 0 if d is of infinite order
ratcurve(d, rc) ==
mn := minIndex(nm := numer(i := minimize ideal d))
h := pmod lift(hh := nm(mn + 1))
b := rmod(retract(retract(nm.mn)@QF)@UP)
s := separate(rmod(retract(norm hh)@UP), b).primePart
bd := badNum rmod denom i
r := resultant(s, b)
bad := lcm [rc.disc, numer r, denom r, bd.den*bd.gcdnum, badNum h]$List(Z)
pp := Q2UPUP(rc.ncurve)
n := rat(pp, d, p := getGoodPrime bad)
-- if n > 1 then it is cheaper to compute the order modulo a second prime,
-- since computing n * d could be very expensive
-- one? n => n
  (n = 1) => n
n := rat(pp, d, getGoodPrime(p * bad))
n = m => n
0

-- returns the order of d mod p
rat(p, d, p) ==
gf := InnerPrimeField p
order(d, pp,
  (q1:F):gf +-> (qq := qmod q1;numer(qq) :: gf / denom(qq) :: gf)
)$ReducedDivisor(F, UP, UPUP, R, gf)

-- returns the potential order of d, 0 if d is of infinite order
possibleOrder d ==
-- zero?(genus()) or one?(#(numer ideal d)) => 1
zero?(genus()) or (#(numer ideal d) = 1) => 1
empty?(la := alglist d) => ratcurve(d, selectIntegers())
not(empty? rest la) =>
  error "PFO::possibleOrder: more than 1 algebraic constant"
algcurve(d, selectIntegers first la, first la)
selectIntegers():RC0 ==
f := definingPolynomial()$R
while zero?(d := doubleDisc(r := polyred pmod f)) repeat newReduc()$q
[r, d]

selectIntegers(k:K):REC ==
g := polyred(f := definingPolynomial()$R)
p := minPoly k
while zero?(d := doubleDisc(r := kpmod(g, k))) or (notIrr? fmod p)
repeat newReduc()$q
[r, d, splitDenominator(fmod p).num]

toQ1(p, d) ==
map((p1:UPQ):F +-> Q2F(retract(p1 rem d)@Q),
p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)
toQ2(p, d) ==
reduce map((p1:UP2):QF +-> toQ1(p1, d)::QF,
p)$UnivariatePolynomialCategoryFunctions2(UP2, UP3, QF, UPUP)

kpmod(p, k) ==
map((p1:QF):UP2 +-> krmod(retract(p1)@UP, k),
p)$UnivariatePolynomialCategoryFunctions2(QF, UPUP, UP2, UP3)

order d ==
zero?(n := possibleOrder(d := reduce d)) => "failed"
principal? reduce(n::Z * d) => n
"failed"

kgetGoodPrime(rec, res, h, b, d) ==
p:PI := 3
while (u := goodRed(rec, res, h, b, d, p)) case "failed" repeat
  p := nextPrime(p::Z)::PI
[p, u::UPQ]
goodRed(rec, res, h, b, d, p) ==
zero?(rec.disc rem p) => "failed"
gf := InnerPrimeField p
l := [f.factor for f in factors
  factor(map((z1:Q):gf +-> retract(z1)@Z :: gf, rec.dfpoly)$SparseUnivariatePolynomialFunctions2(Q, gf))$DistinctDegreeFactorize(gf, SparseUnivariatePolynomial gf) | one?(f.exponent)]
  SparseUnivariatePolynomial gf) | (f.exponent = 1)]
empty? l => "failed"
mdg := first l
for ff in rest l repeat
  if degree(ff) < degree(mdg) then mdg := ff
md := map((z1:gf):Q +-> convert(z1)@Z :: Q, mdg)$SparseUnivariatePolynomialFunctions2(gf, Q)
good?(res, h, b, d, p, md) => md
"failed"

good?(res, h, b, d, p, m) ==
bd := badNum(res rem m)
not (zero?(bd.den rem p) or zero?(bd.gcdnum rem p) or
zero?(kbadBadNum(b,m) rem p) or zero?(kbadBadNum(d,m) rem p)
  or zero?(kbad3Num(h, m) rem p))

——

— PFO.dotabb —

"PFO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFO"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"PFO" -> "FS"
"PFO" -> "FFCAT"

——

package PFOQ PointsOfFiniteOrderRational

— PointsOfFiniteOrderRational.input —

)set break resume
)sys rm -f PointsOfFiniteOrderRational.output
)spool PointsOfFiniteOrderRational.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointsOfFiniteOrderRational
--E 1

)spool
)lisp (bye)

——

— PointsOfFiniteOrderRational.help —

====================================================================
PointsOfFiniteOrderRational examples
==============================================

This package provides function for testing whether a divisor on a curve is a torsion divisor.

See Also:
o )show PointsOfFiniteOrderRational

---

PointsOfFiniteOrderRational (PFOQ)

Exports:
order torsion? torsionIfCan

— package PFOQ PointsOfFiniteOrderRational —

)abbrev package PFOQ PointsOfFiniteOrderRational
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1988
++ Date Last Updated: 3 August 1993
++ Description:
++ This package provides function for testing whether a divisor on a curve is a torsion divisor.

PointsOfFiniteOrderRational(UP, UPUP, R): Exports == Implementation where
UP : UnivariatePolynomialCategory Fraction Integer
UPUP : UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(Fraction Integer, UP, UPUP)
PI ==> PositiveInteger
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Integer
FD ==> FiniteDivisor(Q, UP, UPUP, R)

Exports ==> with
  order : FD -> Union(N, "failed")
  ++ order(f) \ undocumented
  torsion? : FD -> Boolean
  ++ torsion?(f) \ undocumented
  torsionIfCan: FD -> Union(Record(order:N, function:R), "failed")
  ++ torsionIfCan(f) \ undocumented

Implementation ==> add
  import PointsOfFiniteOrderTools(UP, UPUP)

possibleOrder: FD -> N
ratcurve : (FD, UPUP, Z) -> N
rat : (UPUP, FD, PI) -> N

torsion? d == order(d) case N

-- returns the potential order of d, 0 if d is of infinite order
ratcurve(d, modulus, disc) ==
  mn := minIndex(nm := numer(i := ideal d))
  h := lift(hh := nm(mn + 1))
  s := separate(retract(norm hh)@UP,
    b := retract(retract(nm.mn)@Fraction(UP))@UP).primePart
  bd := badNum denom i
  r := resultant(s, b)
  bad := lcm [disc, numer r, denom r, bd.den * bd.gcdnum, badNum h]$List(Z)
  n := rat(modulus, d, p := getGoodPrime bad)
  -- if n > 1 then it is cheaper to compute the order modulo a second prime,
  -- since computing n * d could be very expensive
  -- one? n => n
  (n = 1) => n
  m := rat(modulus, d, getGoodPrime(p * bad))
  n = m => n
  0

rat(pp, d, p) ==
  gf := InnerPrimeField p
  order(d, pp,
    (z1:Q):gf +->
      numer(z1)::gf / denom(z1)::gf)$ReducedDivisor(Q, UP, UPUP, R, gf)

-- returns the potential order of d, 0 if d is of infinite order
possibleOrder d ==
  zero?(genus()) or one?(#(numer ideal d)) => 1
  zero?(genus()) or (#(numer ideal d) = 1) => 1
  r := polyred definingPolynomial()$R
ratcurve(d, r, doubleDisc r)

order d ==
  zero?(n := possibleOrder(d := reduce d)) => "failed"
  principal? reduce(n::Z * d) => n
  "failed"

torsionIfCan d ==
  zero?(n := possibleOrder(d := reduce d)) => "failed"
  (g := generator reduce(n::Z * d)) case "failed" => "failed"
  [n, g::R]

— PFOQ.dotabb —

"PFOQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFOQ"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"PFOQ" -> "FFCAT"

package PFOTOOLS PointsOfFiniteOrderTools

— PointsOfFiniteOrderTools.input —

)set break resume
)sys rm -f PointsOfFiniteOrderTools.output
)spool PointsOfFiniteOrderTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointsOfFiniteOrderTools
--E 1

)spool
)lisp (bye)

— PointsOfFiniteOrderTools.help —
PointsOfFiniteOrderTools examples
====================================================================
Utilities for PFOQ and PFO
See Also:
o )show PointsOfFiniteOrderTools

PointsOfFiniteOrderTools (PFOTOOLS)

--- package PFOTOOLS PointsOfFiniteOrderTools ---

Exports:
badNum doubleDisc getGoodPrime mix polyred

)abbrev package PFOTOOLS PointsOfFiniteOrderTools
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Utilities for PFOQ and PFO

PointsOfFiniteOrderTools(UP, UPUP): Exports == Implementation where
  UP : UnivariatePolynomialCategory Fraction Integer
  UPUP : UnivariatePolynomialCategory Fraction UP

  PI  ==> PositiveInteger
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Integer
**Exports ==> with**

getGoodPrime : Z -> PI
  ++ getGoodPrime n returns the smallest prime not dividing n
badNum : UP -> Record(den:Z, gcdnum:Z)
  ++ badNum(p) \undocumented
badNum : UPUP -> Z
  ++ badNum(u) \undocumented
mix : List Record(den:Z, gcdnum:Z) -> Z
  ++ mix(l) \undocumented
doubleDisc : UPUP -> Z
  ++ doubleDisc(u) \undocumented
polyred : UPUP -> UPUP
  ++ polyred(u) \undocumented

**Implementation ==> add**

import IntegerPrimesPackage(Z)
import UnivariatePolynomialCommonDenominator(Z, Q, UP)

mix l == lcm(lcm [p.den for p in l], gcd [p.gcdnum for p in l])
badNum(p:UPUP) == mix [badNum(retract(c)@UP) for c in coefficients p]
polyred r ==
lcm [commonDenominator(retract(c)@UP) for c in coefficients r] * r
badNum(p:UP) ==
cd := splitDenominator p
[cd.den, gcd [retract(c)@Z for c in coefficients(cd.num)]]

getGoodPrime n ==
p:PI := 3
while zero?(n rem p) repeat
  p := nextPrime(p::Z)::PI
p
doubleDisc r ==
d := retract(discriminant r)@UP
retract(discriminant((d exquo gcd(d, differentiate d))::UP))@Z

———

— PFOTOOLS.dotabb ——

"PFOTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFOTOOLS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFOTOOLS" -> "PFECAT"
package PLPKCRV PolynomialPackageForCurve

— PolynomialPackageForCurve.input —

)set break resume
)set sys rm -f PolynomialPackageForCurve.output
)set spool PolynomialPackageForCurve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialPackageForCurve
--R
--R PolynomialPackageForCurve(K: Field,PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(dim,NonNegativeInteger),dim: NonNegativeInteger,ProjPt: ProjectiveSpaceCategory(K)) is a package constructor
--R Abbreviation for PolynomialPackageForCurve is PLPKCRV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLPKCRV
--R
--R------------------------------- Operations --------------------------------
--R eval : (PolyRing,ProjPt) -> K
--R minimalForm : (PolyRing,ProjPt) -> PolyRing
--R minimalForm : (PolyRing,ProjPt,Integer) -> PolyRing
--R multiplicity : (PolyRing,ProjPt) -> NonNegativeInteger
--R multiplicity : (PolyRing,ProjPt,Integer) -> NonNegativeInteger
--R pointInIdeal? : (List(PolyRing),ProjPt) -> Boolean
--R translateToOrigin : (PolyRing,ProjPt,Integer) -> PolyRing
--R translateToOrigin : (PolyRing,ProjPt) -> PolyRing
--R
--E 1

)spool
)lisp (bye)

——

— PolynomialPackageForCurve.help —

====================================================================
PolynomialPackageForCurve examples
====================================================================

The following is part of the PAFF package

See Also:
o )show PolynomialPackageForCurve
PolynomialPackageForCurve (PLPKCRV)

---

PolynomialPackageForCurve (PLPKCRV)

<table>
<thead>
<tr>
<th>PLPKCRV</th>
</tr>
</thead>
</table>

DIRPCAT

Exports:

eval minimalForm multiplicity pointInIdeal? translateToOrigin

— package PLPKCRV PolynomialPackageForCurve —

)abbrev package PLPKCRV PolynomialPackageForCurve
+Author: Gaetan Hache
+Date Created: 17 nov 1992
+Date Last Updated: May 2010 by Tim Daly
+Description:
+The following is part of the PAFF package
PolynomialPackageForCurve(K,PolyRing,E,dim,ProjPt):Exp == Impl where
K:Field
dim:NonNegativeInteger
E : DirectProductCategory(dim,NonNegativeInteger)
PolyRing: FiniteAbelianMonoidRing(K,E) -- PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)

PackPoly ==> PackageForPoly(K,PolyRing,E,dim)

Exp ==> with

pointInIdeal?: (List(PolyRing),ProjPt) -> Boolean
++ pointInIdeal? test if the given point is in the algebraic set
++ defined by the given list of polynomials.

eval: (PolyRing,ProjPt) -> K
++ eval returns the value at given point.

translateToOrigin: (PolyRing,ProjPt,Integer) -> PolyRing
++ translateToOrigin translate the polynomial from the given point ++ to the origin

translateToOrigin: (PolyRing,ProjPt) -> PolyRing
++ translateToOrigin translate the polynomial from the given point ++ to the origin

minimalForm: (PolyRing,ProjPt) -> PolyRing
++ minimalForm returns the minimal form after translation to ++ the origin.

minimalForm: (PolyRing,ProjPt,Integer) -> PolyRing
++ minimalForm returns the minimal form after translation to ++ the origin.

multiplicity: (PolyRing,ProjPt) -> NonNegativeInteger
++ multiplicity returns the multiplicity of the polynomial at ++ given point.

multiplicity: (PolyRing,ProjPt,Integer) -> NonNegativeInteger
++ multiplicity returns the multiplicity of the polynomial at ++ given point.

Impl ==> add

import PolyRing
import ProjPt
import PackPoly

translateToOrigin(pol,pt,nV)==
zero?(pt.nV) => error "Impossible de translater"
pt:=homogenize(pt,nV)
lpt:List K:=list(pt)$ProjPt
translate(pol,lpt,nV)

pointInIdeal?(lstPol,pt)==
temp:Boolean:=true()$Boolean
for pol in lstPol repeat
temp:=(zero?(eval(pol,pt)) and temp)
temp

eval(f,pt)==
zero? f => 0
lpt:List(K) := list pt
dd:List NonNegativeInteger := entries degree f
lc:= leadingCoefficient f
ee:= reduce( "*" , [ p**e for p in lpt for e in dd | ^zero?(e)], 1$K)
lc * ee + eval( reductum f, pt)

translateToOrigin(pol,pt)==
translateToOrigin(pol,pt,lastNonNull(pt))

Multiplicity(crb,pt) ==
  degreeOfMinimalForm(translateToOrigin(crb,pt))

Multiplicity(crb,pt,nV) ==
  degreeOfMinimalForm(translateToOrigin(crb,pt,nV))

MinimalForm(crb,pt) ==
  minimalForm(translateToOrigin(crb,pt))

MinimalForm(crb,pt,nV) ==
  minimalForm(translateToOrigin(crb,pt,nV))

— PLPKCRV.dotabb —

"PLPKCRV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLPKCRV"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PLPKCRV" -> "DIRPCAT"

— package POLTOPOL PolToPol —

— PolToPol.input —

)set break resume
/sys rm -f PolToPol.output
/spool PolToPol.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolToPol
--E 1

)spool
)lisp (bye)

—
PolToPol (POLTOPOL)

Exports:
  dmpToHdmp  dmpToP  hdmpToDmp  hdmpToP  pToDmp  pToHdmp

— PolToPol.help —

====================================================================
PolToPol examples
====================================================================

Package with the conversion functions among different kind of polynomials

See Also:
o  )show PolToPol

——

)abbrev package POLTOPOL PolToPol
++ Author: P.Gianni, Summer '88
++ Description:
++ Package with the conversion functions among different kind of polynomials

PolToPol(lv,R) : C == T

where
  R      : Ring
  lv     : List Symbol
  NNI    ==> NonNegativeInteger
  Ov     ==> OrderedVariableList(lv)
IES ==> IndexedExponents Symbol
DP ==> DirectProduct(#lv,NonNegativeInteger)
DPoly ==> DistributedMultivariatePolynomial(lv,R)
HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,R)
P ==> Polynomial R
VV ==> Vector NNI
MPC3 ==> MPolyCatFunctions3

C == with
   dmpToHdmp : DPoly -> HDPoly
      ++ dmpToHdmp(p) converts p from a \spadtype{DMP} to a \spadtype{HDMP}.
   hdmpToDmp : HDPoly -> DPoly
      ++ hdmpToDmp(p) converts p from a \spadtype{HDMP} to a \spadtype{DMP}.
   pToHdmp : P -> HDPoly
      ++ pToHdmp(p) converts p from a \spadtype{POLY} to a \spadtype{HDMP}.
   hdmpToP : HDPoly -> P
      ++ hdmpToP(p) converts p from a \spadtype{HDMP} to a \spadtype{POLY}.
   dmpToP : DPoly -> P
      ++ dmpToP(p) converts p from a \spadtype{DMP} to a \spadtype{POLY}.
   pToDmp : P -> DPoly
      ++ pToDmp(p) converts p from a \spadtype{POLY} to a \spadtype{DMP}.

T == add
   variable1(xx:Symbol):Ov == variable(xx)::Ov

   -- transform a P in a HDPoly --
   pToHdmp(pol:P) : HDPoly ==
      map(variable1,pol)$MPC3(Symbol,Ov,IES,HDP,R,P,HDPoly)

   -- transform an HDPoly in a P --
   hdmpToP(hdpol:HDPoly) : P ==
      map(convert,hdpol)$MPC3(Ov,Symbol,HDP,IES,R,HDPoly,P)

   -- transform a DPoly in a P --
   dmpToP(dpol:DPoly) : P ==
      map(convert,dpol)$MPC3(0v,Symbol,DP,IES,R,DPoly,P)

   -- transform a P in a DPoly --
   pToDmp(pol:P) : DPoly ==
      map(variable1,pol)$MPC3(Symbol,0v,IES,DP,R,P,DPoly)

   -- transform a DPoly in a HDPoly --
   dmpToHdmp(dpol:DPoly) : HDPoly ==
      dpol=0 => 0$HDPoly
      monomial(leadingCoefficient dpol, directProduct(degree(dpol)::VV)$HDP)$HDPoly+
         dmpToHdmp(reductum dpol)
-- transform a HDPoly in a DPoly --
hdmpToDmp(hdpol: HDPoly) : DPoly ==
hdpol=0 => 0$DPoly
dd:DP:= directProduct((degree hdpol)::VV)$DP
monomial(leadingCoefficient hdpol,dd)$DPoly+
hdmpToDmp(reductum hdpol)

----

— POLTOPOL.dotabb —

"POLTOPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLTOPOL"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"POLTOPOL" -> "DIRPCAT"

----

package PGROEB PolyGroebner

— PolyGroebner.input —

)set break resume
)sys rm -f PolyGroebner.output
)spool PolyGroebner.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show PolyGroebner
-- E 1

)spool
)lisp (bye)

----

— PolyGroebner.help —

====================================================================
PolyGroebner examples
====================================================================
This package is an interface package to the groebner basis package which allows you to compute groebner bases for polynomials in either lexicographic ordering or total degree ordering refined by reverse lex. The input is the ordinary polynomial type which is internally converted to a type with the required ordering. The resulting groebner basis is converted back to ordinary polynomials. The ordering among the variables is controlled by an explicit list of variables which is passed as a second argument. The coefficient domain is allowed to be any gcd domain, but the groebner basis is computed as if the polynomials were over a field.

See Also:
o )show PolyGroebner

---

PolyGroebner (PGROEB)

Exports:
lexGroebner totalGroebner

--- package PGROEB PolyGroebner ---

)abbrev package PGROEB PolyGroebner
++ Author: P. Gianni
++ Date Created: Summer 1988
++ Description:
++ Groebner functions for P F
++ This package is an interface package to the groebner basis
++ package which allows you to compute groebner bases for polynomials
++ in either lexicographic ordering or total degree ordering refined
++ by reverse lex. The input is the ordinary polynomial type which
++ is internally converted to a type with the required ordering.
++ The resulting groebner basis is converted back to ordinary polynomials.
++ The ordering among the variables is controlled by an explicit list
++ of variables which is passed as a second argument. The coefficient
++ domain is allowed to be any gcd domain, but the groebner basis is
++ computed as if the polynomials were over a field.

PolyGroebner(F) : C == T

where

F  : GcdDomain
NNI ==> NonNegativeInteger
P   ==> Polynomial F
L   ==> List
E   ==> Symbol

C == with
  lexGroebner : (L P,L E) -> L P
    ++ lexGroebner(lp,lv) computes Groebner basis
    ++ for the list of polynomials lp in lexicographic order.
    ++ The variables are ordered by their position in the list lv.

  totalGroebner : (L P, L E) -> L P
    ++ totalGroebner(lp,lv) computes Groebner basis
    ++ for the list of polynomials lp with the terms
    ++ ordered first by total degree and then
    ++ refined by reverse lexicographic ordering.
    ++ The variables are ordered by their position in the list lv.

T == add
  lexGroebner(lp: L P,lv:L E) : L P ==
    PP:= PolToPol(lv,F)
    DPoly := DistributedMultivariatePolynomial(lv,F)
    DP:=DirectProduct(#lv,NNI)
    OV:=OrderedVariableList lv
    b:L DPoly:=\{pToDmp(pol)$PP for pol in lp\}
    gb:L DPoly :=groebner(b)$GroebnerPackage(F,DP,OV,DPoly)
    [dmpToP(pp)$PP for pp in gb]

  totalGroebner(lp: L P,lv:L E) : L P ==
    PP:= PolToPol(lv,F)
    HDPoly := HomogeneousDistributedMultivariatePolynomial(lv,F)
    HDP:=HomogeneousDirectProduct(#lv,NNI)
    OV:=OrderedVariableList lv
    b:L HDPoly:=\{pToHdmp(pol)$PP for pol in lp\}
    gb:=groebner(b)$GroebnerPackage(F,HDP,OV,HDPoly)
    [hdmpToP(pp)$PP for pp in gb]

___
package PAN2EXPR PolynomialAN2Expression

--- PolynomialAN2Expression.input ---

)set break resume
)sys rm -f PolynomialAN2Expression.output
)spool PolynomialAN2Expression.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialAN2Expression
--E 1

)spool
)lisp (bye)

--- PolynomialAN2Expression.help ---

====================================================================
PolynomialAN2Expression examples
====================================================================

This package provides a coerce from polynomials over algebraic numbers to Expression AlgebraicNumber.

See Also:
  o )show PolynomialAN2Expression
PolynomialAN2Expression (PAN2EXPR)

Exports:
  coerce

— package PAN2EXPR PolynomialAN2Expression —

)`abbrev package PAN2EXPR PolynomialAN2Expression
++ Author: Barry Trager
++ Date Created: 8 Oct 1991
++ Description:
++ This package provides a coerce from polynomials over
++ algebraic numbers to \spadtype{Expression AlgebraicNumber}.

PolynomialAN2Expression():Target == Implementation where
  EXPR ==> Expression(Integer)
  AN ==> AlgebraicNumber
  PAN ==> Polynomial AN
  SY ==> Symbol
  Target ==>
    coerce: Polynomial AlgebraicNumber -> Expression(Integer)
      ++ coerce(p) converts the polynomial \spad{p} with algebraic number
      ++ coefficients to \spadtype{Expression Integer}.
    coerce: Fraction Polynomial AlgebraicNumber -> Expression(Integer)
      ++ coerce(rf) converts \spad{rf}, a fraction of polynomial
      ++ \spad{p} with
      ++ algebraic number coefficients to \spadtype{Expression Integer}.
  Implementation ==> add
  coerce(p:PAN):EXPR ==
    map(x+->x::EXPR, y+->y::EXPR, p)$PolynomialCategoryLifting(
      IndexedExponents SY, SY, AN, PAN, EXPR)
  coerce(rf:Fraction PAN):EXPR ==
    numer(rf)::EXPR / denom(rf)::EXPR
package POLYLIFT PolynomialCategoryLifting

-- PolynomialCategoryLifting.input --

)set break resume
)sys rm -f PolynomialCategoryLifting.output
)spool PolynomialCategoryLifting.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialCategoryLifting
--E 1

)spool
)lisp (bye)

---

--- PolynomialCategoryLifting.help ---

====================================================================
PolynomialCategoryLifting examples
====================================================================

This package provides a very general map function, which given a set \( S \) and polynomials over \( \mathbb{R} \) with maps from the variables into \( S \) and the coefficients into \( S \), maps polynomials into \( S \). \( S \) is assumed to support +, *, and **.

See Also:
o )show PolynomialCategoryLifting
PolynomialCategoryLifting (POLYLIFT)

Exports:
map

— package POLYLIFT PolynomialCategoryLifting —

)abbrev package POLYLIFT PolynomialCategoryLifting
++ Author: Manuel Bronstein
++ Description:
++ This package provides a very general map function, which
++ given a set S and polynomials over R with maps from the
++ variables into S and the coefficients into S, maps polynomials
++ into S. S is assumed to support \texttt{+}, \texttt{*} and \texttt{**}.

PolynomialCategoryLifting(E,Vars,R,P,S): Exports == Implementation where
  E : OrderedAbelianMonoidSup
  Vars: OrderedSet
  R : Ring
  P : PolynomialCategory(R, E, Vars)
  S : SetCategory with
      "+" : (%, %) -> %
      "*" : (%, %) -> %
      "**": (%, NonNegativeInteger) -> %

Exports ==> with
  map: (Vars -> S, R -> S, P) -> S
    ++ map(varmap, coefmap, p) takes a
    ++ varmap, a mapping from the variables of polynomial p into S,
    ++ coefmap, a mapping from coefficients of p into S, and p, and
    ++ produces a member of S using the corresponding arithmetic.
    ++ in S
Implementation ==> add
map(fv, fc, p) ==
  (x1 := mainVariable p) case "failed" => fc leadingCoefficient p
  up := univariate(p, x1::Vars)
  t := fv(x1::Vars)
  ans:= fc 0
  while not ground? up repeat
    ans := ans + map(fv,fc, leadingCoefficient up) * t ** (degree up)
    up := reductum up
  ans + map(fv, fc, leadingCoefficient up)

— POLYLIFT.dotabb —

"POLYLIFT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYLIFT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYLIFT" -> "PFECAT"

package POLYCATQ PolynomialCategoryQuotientFunctions

— PolynomialCategoryQuotientFunctions.input —

)set break resume
/sys rm -f PolynomialCategoryQuotientFunctions.output
/spool PolynomialCategoryQuotientFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialCategoryQuotientFunctions
--E 1

)spool
)lisp (bye)

— PolynomialCategoryQuotientFunctions.help —
This package transforms multivariate polynomials or fractions into univariate polynomials or fractions, and back.

See Also:
o )show PolynomialCategoryQuotientFunctions

---

PolynomialCategoryQuotientFunctions (POLYCATQ)

Exports:
variables isExpt isPlus isPower isTimes
mainVariable multivariate univariate

—— package POLYCATQ PolynomialCategoryQuotientFunctions ——

)abbrev package POLYCATQ PolynomialCategoryQuotientFunctions
++ Author: Manuel Bronstein
++ Date Created: March 1988
++ Date Last Updated: 9 July 1990
++ Description:
++ Manipulations on polynomial quotients
++ This package transforms multivariate polynomials or fractions into
++ univariate polynomials or fractions, and back.

PolynomialCategoryQuotientFunctions(E, V, R, P, F):
Exports == Implementation where
E: OrderedAbelianMonoidSup
V: OrderedSet
R: Ring
P: PolynomialCategory(R, E, V)
F: Field with
   coerce: P -> %
   numer : % -> P
   denom : % -> P

UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP

Exports ==> with
   variables : F -> List V
      ++ variables(f) returns the list of variables appearing
      ++ in the numerator or the denominator of f.
   mainVariable: F -> Union(V, "failed")
      ++ mainVariable(f) returns the highest variable appearing
      ++ in the numerator or the denominator of f, "failed" if
      ++ f has no variables.
   univariate : (F, V) -> RF
      ++ univariate(f, v) returns f viewed as a univariate
      ++ rational function in v.
   multivariate: (RF, V) -> F
      ++ multivariate(f, v) applies both the numerator and
      ++ denominator of f to v.
   univariate : (F, V, UP) -> UP
      ++ univariate(f, x, p) returns f viewed as a univariate
      ++ polynomial in x, using the side-condition \spad{p(x) = 0}.
   isPlus : F -> Union(List F, "failed")
      ++ isPlus(p) returns \[m1,\ldots,mn\] if \spad{p = m1 + \ldots + mn} and
      ++ \spad{n > 1}, "failed" otherwise.
   isTimes : F -> Union(List F, "failed")
      ++ isTimes(p) returns \spad{[a1,\ldots,an]} if
      ++ \spad{p = a1 \ldots an} and \spad{n > 1},
      ++ "failed" otherwise.
   isExpt : F -> Union(Record(var:V, exponent:Integer), "failed")
      ++ isExpt(p) returns \spad{[x, n]} if \spad{p = x**n} and \spad{n <> 0},
      ++ "failed" otherwise.
   isPower : F -> Union(Record(val:F, exponent:Integer), "failed")
      ++ isPower(p) returns \spad{[x, n]} if \spad{p = x**n} and \spad{n <> 0},
      ++ "failed" otherwise.

Implementation ==> add
P2UP: (P, V) -> UP

univariate(f, x) == P2UP(numer f, x) / P2UP(denom f, x)
univariate(f, x, modulus) ==
   (bc := extendedEuclidean(P2UP(denom f, x), modulus, 1))
   case "failed" => error "univariate: denominator is 0 mod p"
   (P2UP(numer f, x) * bc.coef1) rem modulus
multivariate(f, x) ==
  v := x::P::F
  ((numer f) v) / ((denom f) v)

mymerge:(List V, List V) -> List V
mymerge(l: List V, m: List V): List V ==
  empty? l => m
  empty? m => l
  first l = first m => cons(first l, mymerge(rest l, rest m))
  first l > first m => cons(first l, mymerge(rest l, m))
  cons(first m, mymerge(l, rest m))

variables f ==
  mymerge(variables numer f, variables denom f)

isPower f ==
  (den := denom f) ^= 1 =>
    (numer f ^= 1 => "failed"
    (ur := isExpt den) case "failed" => [den::F, -1]
    r := ur::Record(var: V, exponent: NonNegativeInteger)
    [r.var::P::F, - (r.exponent::Integer)]
  (ur := isExpt numer f) case "failed" => "failed"
  r := ur::Record(var: V, exponent: NonNegativeInteger)
  [r.var::P::F, r.exponent::Integer]

isExpt f ==
  (ur := isExpt numer f) case "failed" =>
  -- one? numer f =>
    (numer f) = 1 =>
      (ur := isExpt denom f) case "failed" => "failed"
      r := ur::Record(var: V, exponent: NonNegativeInteger)
      [r.var::P::F, - (r.exponent::Integer)]
      "failed"
    r := ur::Record(var: V, exponent: NonNegativeInteger)
  -- one? denom f => [r.var, r.exponent::Integer]
    (denom f) = 1 => [r.var, r.exponent::Integer]
    "failed"

isTimes f ==
  t := isTimes(num := numer f)
  l: Union(List F, "failed") :=
  t case "failed" => "failed"
  [x::F for x in t]
  -- one?(den := denom f) => l
  ((den := denom f) = 1) => l
  -- one? num => "failed"
  num = 1 => "failed"
  d := inv(den::F)
  l case "failed" => [num::F, d]
  concat_!(l::List(F), d)
isPlus f ==
denom f ^= 1 => "failed"
(s := isPlus numer f) case "failed" => "failed"
[x::F for x in s]

mainVariable f ==
a := mainVariable numer f
(b := mainVariable denom f) case "failed" => a
a case "failed" => b
max(a::V, b::V)

P2UP(p, x) ==
map(z +-> z::F,
  univariate(p, x))$SparseUnivariatePolynomialFunctions2(P, F)

-----

— POLYCATQ.dotabb —

"POLYCATQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYCATQ"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYCATQ" -> "PFECAT"

-----

package PCOMP PolynomialComposition

— PolynomialComposition.input —

)set break resume
)sys rm -f PolynomialComposition.output
)spool PolynomialComposition.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialComposition
--E 1

)spool
)lisp (bye)
PolynomialComposition (PCOMP)

PCOMP

PFECAT

Exports:
compose

— package PCOMP PolynomialComposition —

)abbrev package PCOMP PolynomialComposition
++ References: Kozen and Landau, Cornell University  TR 86-773
++ Description:
++ Polynomial composition and decomposition functions
++ If f = g o h then g=leftFactor(f,h) and h=rightFactor(f,g)

PolynomialComposition(UP: UnivariatePolynomialCategory(R), R: Ring): with
  compose: (UP, UP) -> UP
  ++ compose(p,q) \undocumented
== add compose(g, h) ==
   r: UP := 0
   while g ^= 0 repeat
      r := leadingCoefficient(g)*h**degree(g) + r
      g := reductum g
   r

——

— PCOMP.dotabb —

"PCOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PCOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PCOMP" -> "PFECAT"

——

package PDECOMP PolynomialDecomposition

— PolynomialDecomposition.input —

)set break resume
)sys rm -f PolynomialDecomposition.output
)spool PolynomialDecomposition.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialDecomposition
--E 1

)spool
)lisp (bye)

——

— PolynomialDecomposition.help —

====================================================================
PolynomialDecomposition examples
====================================================================
Polynomial composition and decomposition functions

If $f = g \circ h$ then $g = \text{leftFactor}(f, h)$ and $h = \text{rightFactor}(f, g)$

See Also:
- %show PolynomialDecomposition

---

**PolynomialDecomposition (PDECOMP)**

Exports:
- decompose
- leftFactor
- rightFactorCandidate

— package PDECOMP PolynomialDecomposition —

)`abbrev package PDECOMP PolynomialDecomposition
++ References:
++ Dexter Kozen and Susan Landau, ‘‘Polynomial Decomposition Algorithms’’
++ Description:
++ Polynomial composition and decomposition functions
++ If $f = g \circ h$ then $g = \text{leftFactor}(f, h)$ and $h = \text{rightFactor}(f, g)$

PolynomialDecomposition(UP, F): PDcat == PDdef where
  F:Field
  UP:UnivariatePolynomialCategory F
  NNI ==> NonNegativeInteger
  LR ==> Record(left: UP, right: UP)

PDcat == with
  decompose: UP -> List UP
++ decompose(up) \undocumented
decompose: (UP, NNI, NNI) -> Union(LR, "failed")
++ decompose(up,m,n) \undocumented
leftFactor: (UP, UP) -> Union(UP, "failed")
++ leftFactor(p,q) \undocumented
rightFactorCandidate: (UP, NNI) -> UP
++ rightFactorCandidate(p,n) \undocumented
PDdef == add
leftFactor(f, h) ==
g: UP := 0
for i in 0.. while f ^= 0 repeat
fr := divide(f, h)
f := fr.quotient; r := fr.remainder
degree r > 0 => return "failed"
g := g + r * monomial(1, i)
g
decompose(f, dg, dh) ==
df := degree f
dg*dh ^= df => "failed"
h := rightFactorCandidate(f, dh)
g := leftFactor(f, h)
g case "failed" => "failed"
[g::UP, h]
decompose f ==
df := degree f
for dh in 2..df-1 | df rem dh = 0 repeat
h := rightFactorCandidate(f, dh)
g := leftFactor(f, h)
g case UP => return
append(decompose(g::UP), decompose h)
[f]
rightFactorCandidate(f, dh) ==
f := f/leadingCoefficient f
df := degree f
dg := df quo dh
h := monomial(1, dh)
for k in 1..dh repeat
hdg := h**dg
c := (coefficient(f,(df-k)::NNI)-coefficient(hdg,(df-k)::NNI))/(dg::F)
h := h + monomial(c, (dh-k)::NNI)
h - monomial(coefficient(h, 0), 0) -- drop constant term

— PDECOMP.dotabb —
package PFBR PolynomialFactorizationByRecursion

--- PolynomialFactorizationByRecursion.input ---

)set break resume
/sys rm -f PolynomialFactorizationByRecursion.output
/spool PolynomialFactorizationByRecursion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialFactorizationByRecursion
--E 1

)spool
)lisp (bye)

---

--- PolynomialFactorizationByRecursion.help ---

====================================================================
PolynomialFactorizationByRecursion examples
====================================================================

PolynomialFactorizationByRecursion(R,E,VarSet,S)
is used for factorization of sparse univariate polynomials over
a domain S of multivariate polynomials over R.

See Also:
o )show PolynomialFactorizationByRecursion

---
PolynomialFactorizationByRecursion (PFBR)

Exports:
randomR  bivariateSLPEBR  factorByRecursion
factorSFBRlcUnit  factorSquareFreeByRecursion  solveLinearPolynomialEquationByRecursion

icio s e de:
++ PolynomialFactorizationByRecursion(R,E,VarSet,S)
++ is used for factorization of sparse univariate polynomials over
++ a domain S of multivariate polynomials over R.

PolynomialFactorizationByRecursion(R,E, VarSet:OrderedSet, S): public ==
private where
  R:PolynomialFactorizationExplicit
  E:OrderedAbelianMonoidSup
  S:PolynomialCategory(R,E,VarSet)
  PI ==> PositiveInteger
  SupR ==> SparseUnivariatePolynomial R
  SupSupR ==> SparseUnivariatePolynomial SupR
  SupS ==> SparseUnivariatePolynomial S
  SupSupS ==> SparseUnivariatePolynomial SupS
  LPEBFS ==> LinearPolynomialEquationByFractions(S)
public == with
  solveLinearPolynomialEquationByRecursion: (List SupS, SupS) ->
    Union(List SupS,"failed")
    ++ returns the list of polynomials \spad{[q1,...,qn]} such that \spad{\sum qi/pi = p / prod pi}, a
    ++ recursion step for solveLinearPolynomialEquation
    ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
    ++ (see \spadfun{solveLinearPolynomialEquation}).
    ++ If no such list of qi exists, then "failed" is returned.
  factorByRecursion: SupS -> Factored SupS
    ++ factorByRecursion(p) factors polynomial p. This function
++ performs the recursion step for factorPolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorPolynomial}).

\spad{factorSquareFreeByRecursion: SupS -> Factored SupS}
++ factorSquareFreeByRecursion(p) returns the square free
++ factorization of p. This functions performs
++ the recursion step for factorSquareFreePolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorSquareFreePolynomial}).

\spad{randomR: -> R -- has to be global, since has alternative definitions}
++ randomR produces a random element of R

\spad{bivariateSLPEB: (List SupS, SupS, VarSet) -> Union(List SupS,"failed")}
++ bivariateSLPEB(lp,p,v) implements
++ the bivariate case of solveLinearPolynomialEquationByRecursion
++ its implementation depends on R

\spad{factorSFBRlcUnit: (List VarSet, SupS) -> Factored SupS}
++ factorSFBRlcUnit(p) returns the square free factorization of
++ polynomial p
++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivariate})
++ in the case where the leading coefficient of p
++ is a unit.

\spad{private == add}

\spad{supR: SparseUnivariatePolynomial R}
\spad{pp: SupS}
\spad{lpolys,factors: List SupS}
\spad{vv:VarSet}
\spad{lvpolys,lvpp: List VarSet}
\spad{r:R}
\spad{lr:List R}

import FactoredFunctionUtilities(SupS)
import FactoredFunctions2(S,SupS)
import FactoredFunctions2(SupR,SupS)
import CommuteUnivariatePolynomialCategory(S,SupS, SupSupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupSupS)
import UnivariatePolynomialCategoryFunctions2(SupS,SupSupS,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,R,SupR)
import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupR,SupSupR)
import UnivariatePolynomialCategoryFunctions2(SupR,SupSupR,S,SupS)

\spad{hensel: (SupS,VarSet,R,List SupS) ->
  Union(Record(fctrs:List SupS),"failed")}

\spad{chooseSLPEViableSubstitutions: (List VarSet,List SupS,SupS) ->
  Record(substnsField:List R,lpolysRField:List SupR,ppRField:SupR)}
  --++ chooseSLPEViableSubstitutions(lv,lp,p) chooses substitutions
  --++ for the variables in first arg (which are all
  --++ the variables that exist) so that the polys in second argument don't
  --++ drop in degree and remain square-free, and third arg doesn't drop
  --++ drop in degree

\spad{chooseFSQViableSubstitutions: (List VarSet,SupS) ->
  Record(substnsField:List R,ppRField:SupR)}
chooseFSQViableSubstitutions(lv, p) chooses substitutions for the variables in first argument (which are all the variables that exist) so that the second argument poly doesn't drop in degree and remains square-free.

raise: SupR -> SupS
lower: SupS -> SupR

SLPEBR: (List SupS, List VarSet, SupS, List VarSet) -> Union(List SupS, "failed")
factorSFBRlcUnitInner: (List VarSet, SupS, R) -> Union(Factored SupS, "failed")

hensel(pp, vv, r, factors) ==
origFactors := factors
totdegree := 0
prodegree := "max" / [degree(u, vv) for u in coefficients pp]
n := 1
prime := vv - r
foundFactors := empty()
while (totdegree <= prodegree) repeat
  pn := prime ** n
  Ecart := (pp - */ factors) exquo pn
  Ecart case "failed" => error "failed lifting in hensel in PFBR"
  zero? Ecart =>
    -- then we have all the factors
    return [append(foundFactors, factors)]
  step := solveLinearPolynomialEquation(origFactors,
    map(z1 +-> eval(z1, vv, r),
    Ecart))
  step case "failed" => return "failed" -- must be a false split
  factors := [a + b * pn for a in factors for b in step]
  for a in factors for c in origFactors repeat
    pp1 := pp exquo a
    pp1 case "failed" => "next"
    pp := pp1
    prodegree := prodegree + "max" / [degree(u, vv) for u in coefficients a]
    factors := remove(a, factors)
    origFactors := remove(c, origFactors)
    foundFactors := [a, :foundFactors]
  if # factors < 2 =>
    return [(empty? factors => foundFactors;
      [pp, :foundFactors])]
  totdegree := +/*["max" / [degree(u, vv)
    for u in coefficients u1] for u1 in factors]
  n := n + 1
  "failed" -- must have been a false split

factorSFBRlcUnitInner(lvpp, pp, r) ==
-- pp is square-free as a Sup, and its coefficients have precisely
-- the variables of lvpp. Furthermore, its LC is a unit
-- returns "failed" if the substitution is bad, else a factorization
ppR:=map(z1 +-> eval(z1,first lvpp,r),pp)
degree ppR < degree pp => "failed"
degree gcd(ppR,differentiate ppR) >0 => "failed"
factors:=
  empty? rest lvpp =>
    fDown:=factorSquareFreePolynomial map(z1 +-> retract(z1)::R,ppR)
    [:raise (unit fDown * factorList(fDown).first.fctr),
    :[raise u.fctr for u in factorList(fDown).rest]]
  fSame:=factorSFBRlcUnit(rest lvpp,ppR)
  [:unit fSame * factorList(fSame).first.fctr,
  :[uu.fctr for uu in factorList(fSame).rest]]
  #factors = 1 => makeFR(1,["irred",pp,1])
  hen:=hensel(pp,first lvpp,r,factors)
  hen case "failed" => "failed"
  makeFR(1,["irred",u,1] for u in hen.fctrs)
if R has StepThrough then
  factorSFBRlcUnit(lvpp,pp) ==
    val:R := init()
    while true repeat
      tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
      not (tempAns case "failed") => return tempAns
      val1:=nextItem val
      val1 case "failed" =>
        error "at this point, we know we have a finite field"
      val:=val1
    else
      factorSFBRlcUnit(lvpp,pp) ==
        val:R := randomR()
        while true repeat
          tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
          not (tempAns case "failed") => return tempAns
          val := randomR()
        if R has random: -> R then
          randomR() => random()
        else randomR() == (random()$Integer)::R
        if R has FiniteFieldCategory then
          bivariateSLPEBR(1polys,pp,v) ==
            1polysR:List SupSupR:=[map(univariate,u) for u in 1polys]
            ppR: SupSupR:=map(univariate,pp)
            ans:=solveLinearPolynomialEquation(1polysR,ppR)$SupR
            ans case "failed" => "failed"
            [map(z1 +-> multivariate(z1,v),w) for w in ans]
          else
            bivariateSLPEBR(1polys,pp,v) ==
              solveLinearPolynomialEquationByFractions(1polys,pp)$LPEBFS
            chooseFSQViableSubstitutions(lvpp,pp) ==
              substns:List R
              ppR: SupR
while true repeat
    substns:=[randomR() for v in lvpp]
    zero? eval(leadingCoefficient pp,lvpp,substns ) => "next"
    ppR:=[map(z1 +-> (retract eval(z1,lvpp,substns))::R,pp) |
        degree gcd(ppR,differentiate ppR)>0 => "next"
    leave
    [substns,ppR]
chooseSLPEViableSubstitutions(lvpolys,lpolys,pp) ==
    substns:List R
    lpolysR:List SupR
    ppR: SupR
    while true repeat
        substns:=[randomR() for v in lvpolys]
        zero? eval(leadingCoefficient pp,lvpolys,substns ) => "next"
        "or"/[zero? eval(leadingCoefficient u,lvpolys,substns)
            for u in lpolys] => "next"
        lpolysR:=[map(z1 +-> (retract eval(z1,lvpolys,substns))::R,u) |
            for u in lpolys]
        uu:=lpolysR
        while not empty? uu repeat
            "or"/[degree(gcd(uu.first,v))>0 for v in uu.rest] => leave
            uu:=rest uu
            not empty? uu => "next"
        leave
        ppR:=[map(z1 +-> (retract eval(z1,lvpolys,substns))::R,pp) |
            [substns,lpolysR,ppR]
        raise(supR) == map(z1 +-> z1:R::S,supR)
        lower(pp) == map(z1 +-> retract(z1)::R,pp)
        SLPEBR(lpolys,lvpolys,pp,lvpp) ==
            not empty? (m:=setDifference(lvpp,lvpolys)) =>
            v:=first m
            lvpp:=remove(v,lvpp)
            pp1:=swap map(z1 +-> univariate(z1,v),pp)
            -- pp1 is mathematically equal to pp, but is in S[z][v]
            -- so we wish to operate on all of its coefficients
            ans:List SupSupS:= [0 for u in lpolys]
            for m in reverse_! monomials pp1 repeat
                ans1:=SLPEBR(lpolys,lvpolys,leadingCoefficient m,lvpp)
                ans1 case "failed" => return "failed"
                d:=degree m
                ans:=[monomial(a1,d)+a for a in ans for a1 in ans1]
            [map(z1 +-> multivariate(z1,v),swap pp1) for pp1 in ans]
            empty? lvpolys =>
            lpolysR:List SupR
            ppR: SupR
            lpolysR:=[map(retract,u) for u in lpolys]
            ppR:=[map(retract,pp)
            ansR:=solveLinearPolynomialEquation(lpolysR,ppR)
            ansR case "failed" => return "failed"
            [map(z1 +-> z1::S,u) for uu in ansR]
cVS := chooseSLPEViableSubstitutions(lvpolys, lpolys, pp)
ansR := solveLinearPolynomialEquation(cVS.lpolysRField, cVS.ppRField)
ansR case "failed" => "failed"
#lvpolys = 1 => bivariateSLPEBR(lpolys, pp, first lvpolys)
solveLinearPolynomialEquationByRecursion(lpolys, pp) ==
  lvpolys := removeDuplicates_!
  concat [ concat [variables z for z in coefficients u]
    for u in lvpolys]
  lvpp := removeDuplicates_!
  concat [variables z for z in coefficients pp]
  SLPEBR(lpolys, lvpolys, pp, lvpp)

factorByRecursion pp ==
  lv:List(VarSet) := removeDuplicates_!
    concat [variables z for z in coefficients pp]
  empty? lv =>
    concat [variables z for z in coefficients pp]
  c := content pp
  unit? c => refine(squareFree pp, factorSquareFreeByRecursion)
  pp := (pp exquo c)::SupS
  mergeFactors(refine(squareFree pp, factorSquareFreeByRecursion),
    map(z1 -> z1:S::SupS, factor(c)$S))

factorSquareFreeByRecursion pp ==
  lv:List(VarSet) := removeDuplicates_!
    concat [variables z for z in coefficients pp]
  empty? lv =>
    concat [variables z for z in coefficients pp]
  unit? (lcpp := leadingCoefficient pp) => factorSFBRlcUnit(lv, pp)
  oldnfact:NonNegativeInteger := 999999
    -- I hope we never have to factor a polynomial
    -- with more than this number of factors
  lcppPow:S
  while true repeat
    cVS := chooseFSQViableSubstitutions(lv, pp)
    factorsR := factorSquareFreePolynomial(cVS.ppRField)
    (nfact := numberOfFactors factorsR) = 1 =>
      return makeFR(1, [["irred", pp, 1]])
    -- OK, force all leading coefficients to be equal to the leading
    -- coefficient of the input
    nfact > oldnfact => "next" -- can't be a good reduction
    oldnfact := nfact
    factors := [(lcpp exquo leadingCoefficient u.fctr)::S * raise u.fctr
      for u in factorList factorsR]
    ppAdjust := (lcppPow:=(lcpp**#(rest factors)) * pp
    lvppList := lv
    OK := true
    for u in lvppList for v in cVS.substnsField repeat
      hen := hensel(ppAdjust, u, v, factors)
hen case "failed" =>
  OK:=false
  "leave"
factors:=hen.fctrs
OK => leave
factors:=[ (lc:=content w;
  lcppPow:=(lcppPow exquo lc)::S;
  (w exquo lc)::SupS)
  for w in factors]
not unit? lcppPow =>
  error "internal error in factorSquareFreeByRecursion"
makeFR((recip lcppPow)::S::SupS,
    ["irred",w,1] for w in factors))

— PFBR.dotabb —
"PFBR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFBR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFBR" -> "PFECAT"

package PFBRU PolynomialFactorizationByRecursionUnivariate

— PolynomialFactorizationByRecursionUnivariate.input —

)set break resume
)sys rm -f PolynomialFactorizationByRecursionUnivariate.output
)spool PolynomialFactorizationByRecursionUnivariate.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PolynomialFactorizationByRecursionUnivariate
--E 1
)
)spool
)lisp (bye)
PolynomialFactorizationByRecursionUnivariate

R is a PolynomialFactorizationExplicit domain,
S is univariate polynomials over R
We are interested in handling SparseUnivariatePolynomials over S, is a variable we shall call z

See Also:
o )show PolynomialFactorizationByRecursionUnivariate

Exports:
randomR factorByRecursion factorSFBRlcUnit factorSquareFreeByRecursion solveLinearPolynomialEquationByRecursion

Package PFBRU PolynomialFactorizationByRecursionUnivariate
++ S, is a variable we shall call z

PolynomialFactorizationByRecursionUnivariate(R, S): public == private where
  R: PolynomialFactorizationExplicit
  S: UnivariatePolynomialCategory(R)
  PI ==> PositiveInteger
  SupR ==> SparseUnivariatePolynomial R
  SupSupR ==> SparseUnivariatePolynomial SupR
  SupS ==> SparseUnivariatePolynomial S
  SupSupS ==> SparseUnivariatePolynomial SupS
  LPEBFS ==> LinearPolynomialEquationByFractions(S)

public == with
  solveLinearPolynomialEquationByRecursion: (List SupS, SupS) ->
    Union(List SupS, "failed")
    ++ \spad{solveLinearPolynomialEquationByRecursion([p1,...,pn],p)}
    ++ returns the list of polynomials \spad{\{q1,...,qn\}}
    ++ such that \spad{\sum qi/pi = p / \prod pi}, a
    ++ recursion step for solveLinearPolynomialEquation
    ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
    ++ (see \spadfun{solveLinearPolynomialEquation}).
    ++ If no such list of qi exists, then "failed" is returned.
  factorByRecursion: SupS -> Factored SupS
    ++ factorByRecursion(p) factors polynomial p. This function
    ++ performs the recursion step for factorPolynomial,
    ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
    ++ (see \spadfun{factorPolynomial}).
  factorSquareFreeByRecursion: SupS -> Factored SupS
    ++ factorSquareFreeByRecursion(p) returns the square free
    ++ factorization of p. This functions performs
    ++ the recursion step for factorSquareFreePolynomial,
    ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
    ++ (see \spadfun{factorSquareFreePolynomial}).
  randomR: -> R -- has to be global, since has alternative definitions
    ++ randomR() produces a random element of R
  factorSFBRlcUnit: (SupS) -> Factored SupS
    ++ factorSFBRlcUnit(p) returns the square free factorization of
    ++ polynomial p
    ++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivariate}
    ++ in the case where the leading coefficient of p
    ++ is a unit.

private == add
  supR: SparseUnivariatePolynomial R
  pp: SupS
  lpolys, factors: List SupS
  r: R
  lr: List R

import FactoredFunctionUtilities(SupS)
import FactoredFunctions2(SupR, SupS)
import FactoredFunctions2(S, SupS)
import UnivariatePolynomialCategoryFunctions2(S, SupS, R, SupR)
import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
-- local function declarations
raise: SupR -> SupS
lower: SupS -> SupR
factorSFBRlcUnitInner: (SupS,R) -> Union(Factored SupS,"failed")
hensel: (SupS,R,List SupS) ->
union(Record(fctrs:List SupS),"failed")
chooseFSQViableSubstitutions: (SupS) ->
Record(substnsField:R,ppRField:SupR)
---++ chooseFSQViableSubstitutions(p), p is a sup
---++ ("sparse univariate polynomial")
---++ over a sup over R, returns a record
---++ \spad{[substnsField: r, ppRField: q]} where r is a substitution point
---++ q is a sup over R so that the (implicit) variable in q
---++ does not drop in degree and remains square-free.
--- here for the moment, until it compiles
--- N.B., we know that R is NOT a FiniteField, since
--- that is meant to have a special implementation, to break the
--- recursion
solveLinearPolynomialEquationByRecursion(lpolys,pp) ==
lhsdeg:"max"/["max"/[degree v for v in coefficients u] for u in lpolys]
rhsdeg:"max"/[degree v for v in coefficients pp]
lhsdeg = 0 =>
lpolySLower:=[lower u for u in lpolys]
answer:List SupS := [0 for u in lpolys]
for i in 0..rhsdeg repeat
  ppx:=map((z1:S):R +-> coefficient(z1,i),pp)
  zero? ppx => "next"
  recAns:= solveLinearPolynomialEquation(lpolySLower,ppx)
  recAns case "failed" => return "failed"
  answer:=[monomial(1,i)$S * raise c + d
    for c in recAns for d in answer]
answer
solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS
-- local function definitions
hensel(pp,r,factors) ==
  -- factors is a relatively prime factorization of pp modulo the ideal
  -- (x-r), with suitably imposed leading coefficients.
  -- This is lifted, without re-combinations, to a factorization
  -- return "failed" if this can't be done
origFactors:=factors
totdegree:Integer:=0
proddegree:Integer:=
  "max"/[degree(u) for u in coefficients pp]
n:PI:=1
pn:=prime:=monomial(1,1) - r::S
foundFactors:List SupS:=empty()
while (totdegree <= proddegree) repeat
  Ecart:=(pp-*/factors) exquo pn
  Ecart case "failed" =>
error "failed lifting in hensel in PFBRU"
zero? Ecart =>
   -- then we have all the factors
   return [append(foundFactors, factors)]
step:=solveLinearPolynomialEquation(origFactors,
   map(z1 +-> elt(z1,r::S),
   Ecart))
step case "failed" => return "failed" -- must be a false split
factors:=[a+b*pn for a in factors for b in step]
for a in factors for c in origFactors repeat
   pp1:= pp exquo a
   pp1 case "failed" => "next"
   pp:=pp1
   proddegree := proddegree - "max"/[degree(u)
   for u in coefficients a]
   factors:=remove(a,factors)
   origFactors:=remove(c,origFactors)
   foundFactors:=[a,:foundFactors]
#factors < 2 =>
   return [(empty? factors => foundFactors;
   [pp,:foundFactors])
totdegree:= +/"max"/[degree(u)
   for u in coefficients u1]
   for u1 in factors]
n:=n+1
pn:=pn*prime
"failed" -- must have been a false split
chooseFSQViableSubstitutions(pp) ==
   substns:R
   ppR: SupR
   while true repeat
      substns:= randomR()
      zero? elt(leadingCoefficient pp,substns ) => "next"
      ppR:=map(z1 +-> elt(z1,substns),pp)
      degree gcd(ppR,differentiate ppR)>0 => "next"
      leave
      [substns,ppR]
   raise(supR) == map(z1 +-> z1:R::S,supR)
   lower(pp) == map(z1 +-> retract(z1):R,pp)
   factorSFBRlcUnitInner(pp,r) ==
      -- pp is square-free as a Sup, but the Up variable occurs.
      -- Furthermore, its LC is a unit
      -- returns "failed" if the substitution is bad, else a factorization
      ppR:=map(z1 +-> elt(z1,r),pp)
      degree ppR < degree pp => "failed"
      degree gcd(ppR,differentiate ppR) >0 => "failed"
      factors:=
      fDown:=factorSquareFreePolynomial ppR
      [raise (unit fDown * factorList(fDown).first.fctr),
       :[raise u.fctr for u in factorList(fDown).rest]]
#factors = 1 => makeFR(1,[["irred",pp,1]])
hen := hensel(pp,r,factors)
hen case "failed" => "failed"
makeFR(1,[["irred",u,1] for u in hen.fctrs])

-- exported function definitions
if R has StepThrough then
factorSFBRlcUnit(pp) ==
val := R := init()
while true repeat
  tempAns := factorSFBRlcUnitInner(pp,val)
  not (tempAns case "failed") => return tempAns
  val1 := nextItem val
  val1 case "failed" =>
    error "at this point, we know we have a finite field"
  val := val1
else
  factorSFBRlcUnit(pp) ==
  val := R := randomR()
  while true repeat
    tempAns := factorSFBRlcUnitInner(pp,val)
    not (tempAns case "failed") => return tempAns
    val := randomR()
if R has StepThrough then
randomCount := R := init()
randomR() ==
  v := nextItem(randomCount)
  v case "failed" =>
    SAY$Lisp "Taking another set of random values"
    randomCount := init()
    randomCount
  randomCount
randomCount := v
randomCount
else if R has random: -> R then
randomR() == random()
else randomR() == (random()$Integer rem 100)::R
factorByRecursion pp ==
  and/[zero? degree u for u in coefficients pp] =>
    map(raise,factorPolynomial lower pp)
c := content pp
unit? c => refine(squareFree pp,factorSquareFreeByRecursion)
pp := (pp exquo c)::SupS
mergeFactors(refine(squareFree pp,factorSquareFreeByRecursion),
  map(z1 +-> z1:S::SupS,factor(c)$S))
factorSquareFreeByRecursion pp ==
  and/[zero? degree u for u in coefficients pp] =>
    map(raise,factorSquareFreePolynomial lower pp)
unit? (lcpp := leadingCoefficient pp) => factorSFBRlcUnit(pp)
oldnfact := NonNegativeInteger := 999999
-- I hope we never have to factor a polynomial
-- with more than this number of factors
begin

while true repeat -- a loop over possible false splits
    -- OK, force all leading coefficients to be equal to the leading
    -- coefficient of the input
    nfact > oldnfact => "next" -- can't be a good reduction

    oldnfact := nfact
    lcppR := leadingCoefficient cVS.ppRF
    factorsR := [raise((lcppR exquo leadingCoefficient u.fctr) ::R * u.fctr)
                 for u in factorList factorsR]
    -- factors now multiplies to give cVS.ppRF * lcppR^(#factors-1)
    -- Now change the leading coefficient to be lcpp
    -- factors := [monomial(lcpp, degree u) + reductum u for u in factors]
    factors := [(lcpp exquo leadingCoefficient u.fctr) ::S * raise u.fctr
                for u in factorList factorsR]
    ppAdjust := (lcppPow := lcpp**(#rest factors)) * pp
    OK := true
    hen := hensel(ppAdjust, cVS.substnsField, factors)
    hen case "failed" => "next"
    factors := hen.fctrs
    leave

factors := [ (lc := content w;
              lcppPow := (lcppPow exquo lc) ::S;
              (w exquo lc) :: SupS)
              for w in factors]
not unit? lcppPow =>
    error "internal error in factorSquareFreeByRecursion"
makeFR((recip lcppPow) :: S :: SupS,
       [["irred", w, 1] for w in factors])

end
package POLY2 PolynomialFunctions2

— PolynomialFunctions2.input —

)set break resume
)sys rm -f PolynomialFunctions2.output
)spool PolynomialFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialFunctions2
--E 1

)spool
)lisp (bye)

— PolynomialFunctions2.help —

====================================================================
PolynomialFunctions2 examples
====================================================================

This package takes a mapping between coefficient rings, and lifts it to a mapping between polynomials over those rings.

See Also:
o )show PolynomialFunctions2

——
PolynomialFunctions2 (POLY2)

Exports:
map

— package POLY2 PolynomialFunctions2 —

)abbrev package POLY2 PolynomialFunctions2
++ Description:
++ This package takes a mapping between coefficient rings, and lifts
++ it to a mapping between polynomials over those rings.

PolynomialFunctions2(R:Ring, S:Ring): with
   map: (R -> S, Polynomial R) -> Polynomial S
   ++ map(f, p) produces a new polynomial as a result of applying
   ++ the function f to every coefficient of the polynomial p.
   == add
   map(f, p) == map(x1 +-> x1::Polynomial(S), x2 +-> f(x2)::Polynomial(S),
   p)$PolynomialCategoryLifting(IndexedExponents Symbol,
   Symbol, R, Polynomial R, Polynomial S)

— POLY2.dotabb —

"POLY2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLY2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLY2" -> "PFECAT"
package PGCD PolynomialGcdPackage

--- PolynomialGcdPackage.input ---

)set break resume
)sys rm -f PolynomialGcdPackage.output
)spool PolynomialGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialGcdPackage
--E 1

)spool
)lisp (bye)

---

--- PolynomialGcdPackage.help ---

====================================================================
PolynomialGcdPackage examples
====================================================================

This package computes multivariate polynomial gcd's using a hensel lifting strategy. The contraint on the coefficient domain is imposed by the lifting strategy. It is assumed that the coefficient domain has the property that almost all specializations preserve the degree of the gcd.

See Also:
o )show PolynomialGcdPackage

---
PolynomialGcdPackage (PGCD)

Exports:
   gcd    gcdPrimitive

The simple Euclidean division algorithm,

given $a \neq 0$, $b \neq 0$ where $a$ and $b$ are two polynomials in the variable $x$, then

output $q$, the quotient and $r$, the remainder

begin
   Let
   $q = 0$
   $r = a$
   $d = \text{degree}(b)$
   $c = \text{leadingCoefficient}(b)$

   while $\text{degree}(r) \geq d$ do
      $e = \text{degree}(r) - d$
      $s = \frac{\text{leadingCoefficient}(r)x^e}{c}$
      $q = q + s$
      $r = r - sb$
   endwhile

   return $(q, r)$
end

For example, given

$p_1 := (x+1) \ast (x+6)$
$p_2 := (x+1) \ast (x-6)$
gcd(p1,p2) -> x+1

— package PGCD PolynomialGcdPackage —
++ Author: Michael Lucks, P. Gianni, Frederic Lehobey
++ Date Last Updated: 17 June 1996
++ Description:
++ This package computes multivariate polynomial gcd's using
++ a hensel lifting strategy. The constraint on the coefficient
++ domain is imposed by the lifting strategy. It is assumed that
++ the coefficient domain has the property that almost all specializations
++ preserve the degree of the gcd.

PolynomialGcdPackage(E,OV,R,P):C == T where
  I ==> Integer
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  R : EuclideanDomain
  P : PolynomialCategory(R,E,OV)
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  SUPP ==> SparseUnivariatePolynomial P

C == with
  gcd : (P,P) -> P
    ++ gcd(p,q) computes the gcd of the two polynomials p and q.
    ++
    ++ X p1:=(x+1)*(x+6)
    ++ X p2:=(x+1)*(x-6)
    ++ X gcd(p1,p2)
  gcd : List P -> P
    ++ gcd(lp) computes the gcd of the list of polynomials lp.
  gcd : (SUPP,SUPP) -> SUPP
    ++ gcd(p,q) computes the gcd of the two polynomials p and q.
  gcd : List SUPP -> SUPP
    ++ gcd(lp) computes the gcd of the list of polynomials lp.
  gcdPrimitive : (P,P) -> P
    ++ gcdPrimitive(p,q) computes the gcd of the primitive polynomials
    ++ p and q.
  gcdPrimitive : (SUPP,SUPP) -> SUPP
    ++ gcdPrimitive(p,q) computes the gcd of the primitive polynomials
    ++ p and q.
  gcdPrimitive : List P -> P
    ++ gcdPrimitive lp computes the gcd of the list of primitive
    ++ polynomials lp.

T == add
  SUPP ==> SparseUnivariatePolynomial R
  LGcd ==> Record(locgcd:SUPP,goodint:List List R)
  UTerm ==> Record(lpol:List SUP,lint:List List R,mpol:SUPP)
  pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer)):R
import MultivariateLifting(E,OV,R,P)
import FactoringUtilities(E,OV,R,P)

-------- Local Functions --------
myran : Integer -> Union(R,"failed")
better : (P,P) -> Boolean
failtest : (SUPP,SUPP,SUPP) -> Boolean
monomContent : (SUPP) -> SUPP
gcdMonom : (SUPP,SUPP) -> SUPP
gcdTermList : (P,P) -> P
good : (SUPP,List OV,List List R) -> Record(upol:SUP,inval:List List R)
chooseVal : (SUPP,SUPP,List OV,List List R) -> Union(UTerm,"failed")
localgcd : (SUPP,SUPP,List OV,List List R) -> LGcd
notCoprime : (SUPP,SUPP, List NNI,List OV,List List R) -> SUPP
imposelc : (List SUP,List OV,List R,List P) ->
            Union(List SUP, "failed")
lift? : (SUPP,SUPP,UTerm,List NNI,List OV) ->
            Union(s:SUPP,failed:"failed",notCoprime:"notCoprime")
lift : (SUPP,SUPP,P,List OV,List NNI,List List R) -> Union(SUPP,"failed")

---- Local functions ----
-- test if something wrong happened in the gcd
failtest(f:SUPP,p1:SUPP,p2:SUPP) : Boolean ==
    (p1 exquo f) case "failed" or (p2 exquo f) case "failed"

-- test if something wrong happened in the gcd
chooseVal(p1:SUPP,p2:SUPP,lvr:List OV,_
         ltry:List List R):Union(UTerm,"failed") ==
    d1:=degree(p1)
    d2:=degree(p2)
    dd:NNI:=0$NNI
    nvr:NNI:=#lvr
    lval:List R :=[]
    range:I:=8
    repeat
        range:=2*range
        lval:=[ran(range) for i in 1..nvr]
        member?(lval,ltry) => "new point"
        ltry:=cons(lval,ltry)
        uf1:SUP:=completeEval(p1,lvr,lval)
        degree uf1 ^= d1 => "new point"
        uf2:SUP:=completeEval(p2,lvr,lval)
        degree uf2 ^= d2 => "new point"
        u:=gcd(uf1,uf2)
        du:=degree u
        --the univariate gcd is 1
if $du=0$ then return $[[1$SUP],ltry,0$SUPP]$UTerm

$ugcd$:List SUP:=[u,(uf1 exquo u)::SUP,(uf2 exquo u)::SUP]
$uterm:=[ugcd,ltry,0$SUPP]$UTerm
$dd=0 => dd:=du$

--the degree is not changed
$du/dd =>$

--test if one of the polynomials is the gcd
$dd=d1 =>$
  if `((f:=p2 exquo p1) case "failed") then
    return $[[u],ltry,p1]$UTerm
  if dd=d2 then dd:=(dd-1)::NNI
$dd=d2 =>$
  if `((f:=p1 exquo p2) case "failed") then
    return $[[u],ltry,p2]$UTerm
  dd:=(dd-1)::NNI
return $uterm$

--the new gcd has degree less
$du/dd => dd:=du$

good(f:SUPP,lvr:List OV, _
  ltry:List List R):Record(upol:SUP,inval:List List R) ==
  nvr:NNI:=#lvr
  range:I:=1
while true repeat
  range:=2*range
  lval:=[ran(range) for i in 1..nvr]
  member?(lval,ltry) => "new point"
  ltry:=cons(lval,ltry)
  uf:=completeEval(f,lvr,lval)
  if degree gcd(uf,differentiate uf)=0 then return [uf,ltry]

In Gathen [?] we find a discussion of applying the Euclidean algorithm to elements of a field. In a field every nonzero rational number is a unit. If we want to define a single element such that

$$gcd(f, g) \in \mathbb{Q}[x]$$

we choose a monic polynomial, that is, the element with a leading coefficient of 1. So

$$abs(a/ leadingCoefficient(a))$$

defines a normal form of $a$.

Gathen also notes that for polynomials we should modify the Euclidean algorithm so all remainders are normalized. Without this constraint the remainders will have huge numerators
and denominators.

-- package PGCD PolynomialGcdPackage --

-- impose the right leading condition, check for failure.
imposelc(lipol:List SUP, lvar:List OV, lval:List R,
leadc:List P): Union(List SUP, "failed") ==
result:List SUP :=[]
for pol in lipol for leadpol in leadc repeat
p1 := univariate eval(leadpol, lvar, lval) * pol
plu := p1 exquo leadingCoefficient pol
plu case "failed" => return "failed"
result := cons(plu::SUP, result)
reverse result

-- Compute the gcd between not coprime polynomials
notCoprime(g:SUPP, p2:SUPP, ldeg:List NNI,_,
lvar1:List OV, ltry:List List R) : SUPP ==
g1:=gcd(g,differentiate g)
l1 := (g exquo g1)::SUPP
lg:LGcd:=localgcd(l1,p2,lvar1,ltry)
(1,ltry):=(lg.locgcd,lg.goodint)
lval:=ltry.first
p2l:=(p2 exquo l)::SUPP
(gd1,gd2):=(1,1)
ul:=[completeEval(1,1,lvar1,lval)
dl:=degree ul
if degree gcd(ul,differentiate ul) ^=0 then
newchoice:=good(1,1,lvar1,ltry)
ul:=newchoice.upol
ltry:=newchoice.inval
lval:=ltry.first
ug1:=completeEval(g1,lvar1,lval)
ulist:=[ug1,completeEval(p2l,lvar1,lval)]
lcpol:List P:=[leadingCoefficient g1, leadingCoefficient p2]
while true repeat
d:SUP:=gcd(cons(ul,ulist))
if degree d =0 then return gd1
lquo:=(ul exquo d)::SUP
if degree lquo ^=0 then
lgcd:=gcd(cons(leadingCoefficient l,lcpol))
(gd1:=lift(1,d,lquo,lgcd,lvar1,ldeg,lval)) case "failed" =>
return notCoprime(g,p2,ldeg,lvar1,ltry)
l:=gd2:=gd1::SUPP
ul:=completeEval(1,lvar1,lval)
dl:=degree ul
gd1:=gd1*gd2
ulist:=[(uf exquo d)::SUP for uf in ulist]
gcdPrimitive(p1:SUPP,p2:SUPP) : SUPP ==
if (d1:=degree(p1)) > (d2:=degree(p2)) then
  (p1,p2):= (p2,p1)
  (d1,d2):= (d2,d1)
degree p1 = 0 =>
  p1 = 0 => unitCanonical p2
  unitCanonical p1
lvar:List OV:=
  sort((a:OV,b:OV):Boolean+->a>b,setUnion(variables p1,variables p2))
empty? lvar =>
  raisePolynomial(gcd(lowerPolynomial p1,lowerPolynomial p2))
(p2 exquo p1) case SUPP => unitCanonical p1
ltry:List List R:=empty()
totResult:=localgcd(p1,p2,lvar,ltry)
result: SUPP:=totResult.locgcd
-- special cases
result=1 => 1$SUPP
while failtest(result,p1,p2) repeat
  -- SAY$Lisp "retrying gcd"
  ltry:=totResult.goodint
  totResult:=localgcd(p1,p2,lvar,ltry)
  result:=totResult.locgcd
result

--local function for the gcd : it returns the evaluation point too
localgcd(p1:SUPP,p2:SUPP,lvar:List(OV),ltry:List List R) : LGcd ==
  uterm:=chooseVal(p1,p2,lvar,ltry)::UTerm
  ltry:=uterm.lint
  listpol:= uterm.lpol
  ud:=listpol.first
  dd:= degree ud

  --the univariate gcd is 1
  dd=0 => [1$SUPP,ltry]$LGcd

  --one of the polynomials is the gcd
  dd=degree(p1) or dd=degree(p2) =>
    [uterm.mpol,ltry]$LGcd
  ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))
  -- if there is a polynomial g s.t. g/gcd and gcd are coprime ...
  -- I can lift
  (h:=lift?(p1,p2,uterm,ldeg,lvar)) case notCoprime =>
    [notCoprime(p1,p2,ldeg,lvar),ltry]$LGcd
  h case failed => localgcd(p1,p2,lvar,ltry) -- skip bad values?
    [h.s,ltry]$LGcd

  -- content, internal functions return the poly if it is a monomial
  monomContent(p:SUPP):SUPP ==
degree(p)=0 \Rightarrow 1
\text{md}:= \text{minimumDegree}(p)
\text{monomial}(\text{gcd sort(better,coefficients } p),\text{md})

-- Ordering for gcd purposes
\text{better}(p1:P,p2:P):\text{Boolean} ==
ground? p1 \Rightarrow true
ground? p2 \Rightarrow false
degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

\text{best_to_front}(l : \text{List } P) : \text{List } P ==
res := []
best := first(l)
for p in rest l repeat
  if \text{better}(p, best) then
    res := cons(best, ress)
    best := p
  else
    res := cons(p, ress)
cons(best, ress)

-- Gcd between polynomial p1 and p2 with
-- mainVariable p1 < x=mainVariable p2
\text{gcdTermList}(p1:P,p2:P) : P ==
termList := \text{best_to_front}(
  cons(p1,\text{coefficients univariate}(p2,(\text{mainVariable } p2)::OV)))
q:P:=termList.first
for term in termList.rest until q = 1$P repeat q:= \text{gcd}(q,term)
q

-- Gcd between polynomials with the same mainVariable
\text{gcd}(p1:\text{SUPP},p2:\text{SUPP}): \text{SUPP} ==
if degree(p1) > degree(p2) then (p1,p2):= (p2,p1)
degree p1 = 0 =>
p1 = 0 => \text{unitCanonical } p2
p1 = 1 => \text{unitCanonical } p1
\text{gcd}(\text{leadingCoefficient } p1, \text{content } p2)::\text{SUPP}
\text{reductum}(p1)=0 => \text{gcdMonom}(p1,\text{monomContent } p2)
c1:= \text{monomContent}(p1)
\text{reductum}(p2)=0 => \text{gcdMonom}(c1,p2)
c2:= \text{monomContent}(p2)
p1:= (p1 \text{ exquo } c1):\text{SUPP}
p2:= (p2 \text{ exquo } c2)::\text{SUPP}
\text{gcdPrimitive}(p1,p2) * \text{gcdMonom}(c1,c2)

-- gcd between 2 monomials
\text{gcdMonom}(m1:\text{SUPP},m2:\text{SUPP}):\text{SUPP} ==
\text{monomial}(\text{gcd}(\text{leadingCoefficient}(m1),\text{leadingCoefficient}(m2)) ,
\min(\text{degree}(m1),\text{degree}(m2)))
Hensel’s lifting lemma states that if a polynomial equation has a simple root modulo a prime number \( p \), then this root corresponds to a unique root of the same equation modulo any higher power of \( p \), which can be found by iteratively “lifting” the solution modulo successive powers of \( p \).

See Volume 10.1 for more details.

---

package PGCD PolynomialGcdPackage ---

-- If there is a pol s.t. \( \text{pol}/\text{gcd} \) and gcd are coprime I can lift
lift?(p1:SUPP,p2:SUPP,uterm:UTerm,ldeg:List NNI, _
  lvar:List OV) : _
  Union(s:SUPP,failed:"failed",notCoprime:"notCoprime") ==
  (listpol, lval) := (uterm.lpol, first(uterm.lint))
  d := first(listpol)
  listpol := rest(listpol)
  uf := listpol(1)
  f := p1
  -- note uf and d not necessarily primitive
  if degree gcd(uf, d) ^= 0 then
    uf := listpol(2)
    f := p2
  if degree gcd(uf, d) ^= 0 then return ["notCoprime"]
  lgcd := gcd(leadingCoefficient p1, leadingCoefficient p2)
  l := lift(f, d, uf, lgcd, lvar, ldeg, lval)
  l case "failed" => ["failed"]
  [l :: SUPP]

-- interface with the general "lifting" function
lift(f:SUPP,d:SUP,uf:SUP,lgcd:P,lvar:List OV,
  ldeg:List NNI,lval:List R):Union(SUPP,"failed") ==
  leadpol : Boolean := false
  lcf : P
  lcf := leadingCoefficient f
  df := degree f
  leadlist : List(P) := []
  if lgcd ^= 1 then
    leadpol := true
    f := lgcd*f
  ldeg :=[n0+n1 for n0 in ldeg for n1 in degree(lgcd, lvar)]
  lcd : R := leadingCoefficient d
  lgcd1 :=
  degree(lgcd) = 0 => retract lgcd
  retract(eval(lgcd, lvar, lval))
  du := (lgcd1*d) exquo lcd
  du case "failed" => "failed"
  d := du::SUP
uf := lcd*uf
leadlist := [lgcd, lcf]
lgu := imposelc([d, uf], lvar, lval, leadlist)
lgu case "failed" => "failed"
lg := lgu::List(SUP)
(pl := lifting(f,lvar,lg,lval,leadlist,ldeg,pmod)) case "failed" => 
"failed"
plist := pl :: List SUP
(p0 : SUPP, p1 : SUPP) := (plist.first, plist.2)
if completeEval(p0, lvar, lval) ^= lg.first then
(p0, p1) := (p1, p0)
not leadpol => p0
p0 exquo content(p0)

-- Gcd for two multivariate polynomials
gcd(p1:P,p2:P) : P ==
ground? p1 =>
p1 := unitCanonical p1
p1 = 1$P => p1
p1 = 0$P => unitCanonical p2
ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
gcdTermList(p1,p2)
ground? p2 =>
p2 := unitCanonical p2
p2 = 1$P => p2
p2 = 0$P => unitCanonical p1
gcdTermList(p2,p1)
(p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
mv1= mainVariable(p1)::OV
mv2= mainVariable(p2)::OV
mv1 < mv2 => multivariate(gcd(univariate(p1,mv1),
univariate(p2,mv1)),mv1)
mv1 < mv2 => gcdTermList(p1,p2)
gcdTermList(p2,p1)

-- Gcd for a list of multivariate polynomials
gcd(listp:List P) : P ==
if := best_to_front(listp)
f:=lf.first
for g in lf.rest repeat
f:=gcd(f,g)
if f=1$P then return f
f

gcd(listp:List SUPP) : SUPP ==
lf:=sort((z1:SUPP,z2:SUPP):Boolean +-> degree(z1)<degree(z2),listp)
f:=lf.first
for g in lf.rest repeat
f:=gcd(f,g)
if f=1 then return f
-- Gcd for primitive polynomials
gcdPrimitive(p1:P,p2:P):P ==
  (p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
  ground? p1 =>
    ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
    p1 = 0$P => p2
    1$P
  ground? p2 =>
    p2 = 0$P => p1
    1$P
mv1:= mainVariable(p1)::OV
mv2:= mainVariable(p2)::OV
mv1 = mv2 =>
  md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv2))
  mp:=1$P
  if md>1 then
    mp:=(mv1::P)**md
    p1:=(p1 exquo mp)::P
    p2:=(p2 exquo mp)::P
  up1 := univariate(p1,mv1)
  up2 := univariate(p2,mv2)
  mp*multivariate(gcdPrimitive(up1,up2),mv1)
  1$P

-- Gcd for a list of primitive multivariate polynomials
gcdPrimitive(listp:List P) : P ==
  lf:=sort(better,listp)
  f:=lf.first
  for g in lf.rest repeat
    f:=gcdPrimitive(f,g)
    if f=1$P then return f
f
package PINTERP PolynomialInterpolation

--- PolynomialInterpolation.input ---

)set break resume
)sys rm -f PolynomialInterpolation.output
)spool PolynomialInterpolation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialInterpolation
--E 1

)spool
)lisp (bye)

---

--- PolynomialInterpolation.help ---

====================================================================
PolynomialInterpolation examples
====================================================================

This package exports interpolation algorithms

See Also:
  o )show PolynomialInterpolation

---
PolynomialInterpolation (PINTERP)

Exports:
interpolate

— package PINTERP PolynomialInterpolation —

)abbrev package PINTERP PolynomialInterpolation
++ Description:
++ This package exports interpolation algorithms

PolynomialInterpolation(xx, F): Cat == Body where
  xx: Symbol
  F: Field
  UP ==> UnivariatePolynomial
  SUP ==> SparseUnivariatePolynomial

Cat ==> with
  interpolate: (UP(xx,F), List F, List F) -> UP(xx,F)
    ++ interpolate(u,lf,lg) undocumented
  interpolate: (List F, List F) -> SUP F
    ++ interpolate(lf,lg) undocumented

Body ==> add
  PIA ==> PolynomialInterpolationAlgorithms

  interpolate(qx, lx, ly) ==
    px := LagrangeInterpolation(lx, ly)$PIA(F, UP(xx, F))
    elt(px, qx)

  interpolate(lx, ly) ==
    LagrangeInterpolation(lx, ly)$PIA(F, SUP F)

________
package PINTERPA PolynomialInterpolationAlgorithms

--- PolynomialInterpolationAlgorithms.input ---

)set break resume
/sys rm -f PolynomialInterpolationAlgorithms.output
/spool PolynomialInterpolationAlgorithms.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show PolynomialInterpolationAlgorithms
-- 1

)spool
)lisp (bye)

--- PolynomialInterpolationAlgorithms.help ---

====================================================================
PolynomialInterpolationAlgorithms examples
====================================================================

This package exports interpolation algorithms

See Also:
  o )show PolynomialInterpolationAlgorithms

---
PolynomialInterpolationAlgorithms (PINTERPA)

Exports:
LagrangeInterpolation

--- package PINTERPA PolynomialInterpolationAlgorithms ---

)abbrev package PINTERPA PolynomialInterpolationAlgorithms
++ Description:
++ This package exports interpolation algorithms

PolynomialInterpolationAlgorithms(F, P): Cat == Body where
F: Field
P: UnivariatePolynomialCategory(F)

Cat ==> with
  LagrangeInterpolation: (List F, List F) -> P
  ++ LagrangeInterpolation(l1,l2) \undocumented

Body ==> add
  LagrangeInterpolation(lx, ly) ==
  #lx ^= #ly =>
    error "Different number of points and values."
  ip: P := 0
  for xi in lx for yi in ly for i in 0.. repeat
    pp: P := 1
    xp: F := 1
    for xj in lx for j in 0.. | i ^= j repeat
      pp := pp * (monomial(1,1) - monomial(xj,0))
      xp := xp * (xi - xj)
    ip := ip + (yi/xp) * pp
  ip

---
package PNTHEORY PolynomialNumberTheoryFunctions

— PolynomialNumberTheoryFunctions.input —

)set break resume
)sys rm -f PolynomialNumberTheoryFunctions.output
)spool PolynomialNumberTheoryFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialNumberTheoryFunctions
--E 1

)spool
)lisp (bye)

— PolynomialNumberTheoryFunctions.help —

====================================================================
PolynomialNumberTheoryFunctions examples
====================================================================

This package provides various polynomial number theoretic functions
over the integers.

See Also:
  o )show PolynomialNumberTheoryFunctions

———
PolynomialNumberTheoryFunctions (PNTHEORY)

Exports:
bernoulli chebyshevT chebyshevU cyclotomic euler
fixedDivisor hermite laguerre legendre

-- package PNTHEORY PolynomialNumberTheoryFunctions --

)abbrev package PNTHEORY PolynomialNumberTheoryFunctions
++ Author: Michael Monagan, Clifton J. Williamson
++ Date Created: June 1987
++ Date Last Updated: 10 November 1996 (Claude Quitte)
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various polynomial number theoretic functions
++ over the integers.

PolynomialNumberTheoryFunctions(): Exports == Implementation where
I ==> Integer
RN ==> Fraction I
SUP ==> SparseUnivariatePolynomial
NNI ==> NonNegativeInteger

Exports ==> with
bernoulli : I -> SUP RN
  ++ bernoulli(n) returns the nth Bernoulli polynomial \spad{B[n](x)}.
  ++ Bernoulli polynomials denoted \spad{B(n,x)} computed by solving the
  ++ differential equation \spad{differentiate(B(n,x),x) = n B(n-1,x)} where
  ++ \spad{B(0,x) = 1} and initial condition comes from \spad{B(n) = B(n,0)}.
chebyshevT: I -> SUP I
  ++ chebyshevT(n) returns the nth Chebyshev polynomial \spad{T[n](x)}.
  ++ Note that Chebyshev polynomials of the first kind,
  ++ denoted \spad{T[n](x)},
  ++ computed from the two term recurrence. The generating function
  ++ \spad{((1-t*x)/(1-2*t*x+t**2)} = sum(\spad{T[n](x)}*t**n, n=0..infinity).
chebyshevU: I -> SUP I
  ++ chebyshevU(n) returns the nth Chebyshev polynomial \spad{U[n](x)}.
++ Note that Chebyshev polynomials of the second kind,
++ denoted \spad{U[n](x)},
++ computed from the two term recurrence. The generating function
\spad{1/(1-2*t*x+t**2) = sum(T[n](x)*t**n, n=0..infinity)}.

\texttt{cyclotomic: I \to SUP I}
++ cyclotomic(n) returns the nth cyclotomic polynomial \spad{phi[n](x)}.
++ Note that \spad{phi[n](x)} is the factor of \spad{x**n - 1} whose roots
++ are the primitive nth roots of unity.

\texttt{euler : I \to SUP RN}
++ euler(n) returns the nth Euler polynomial \spad{E[n](x)}.
++ Note that Euler polynomials denoted \spad{E(n,x)} computed by solving
++ the differential equation
\spad{differentiate(E(n,x),x) = n E(n-1,x)} where
++ \spad{E(0,x) = 1} and initial condition comes
++ from \spad{E(n) = 2**n E(n,1/2)}.

\texttt{fixedDivisor: SUP I \to I}
++ fixedDivisor(a) for \spad{a(x)} in \spad{Z[x]} is the largest integer
++ f such that f divides \spad{a(x=k)} for all integers k.
++ Note that fixed divisor of \spad{a} is
++ \spad{reduce(gcd,[a(x=k) for k in 0..degree(a)])}.

\texttt{hermite : I \to SUP I}
++ hermite(n) returns the nth Hermite polynomial \spad{H[n](x)}.
++ Note that Hermite polynomials, denoted \spad{H[n](x)}, are computed from
++ the two term recurrence. The generating function is:
\spad{exp(2*t*x-t**2) = sum(H[n](x)*t**n/n!, n=0..infinity)}.

\texttt{laguerre : I \to SUP I}
++ laguerre(n) returns the nth Laguerre polynomial \spad{L[n](x)}.
++ Note that Laguerre polynomials, denoted \spad{L[n](x)}, are computed
++ from the two term recurrence. The generating function is:
\spad{exp(x*t/(t-1))/(1-t) = sum(L[n](x)*t**n/n!, n=0..infinity)}.

\texttt{legendre : I \to SUP RN}
++ legendre(n) returns the nth Legendre polynomial \spad{P[n](x)}.
++ Note that Legendre polynomials, denoted \spad{P[n](x)}, are computed
++ from the two term recurrence. The generating function is:
\spad{1/sqrt(1-2*t*x+t**2) = sum(P[n](x)*t**n, n=0..infinity)}.

Implementation ==> add

\texttt{import IntegerPrimesPackage(I)}

\texttt{x := monomial(1,1)$SUP(I)}
\texttt{y := monomial(1,1)$SUP(RN)}

-- For functions computed via a fixed term recurrence we record
-- previous values so that the next value can be computed directly

\texttt{E : Record(En:I, Ev:SUP(RN)) := [0,1]}
\texttt{B : Record( Bn:I, Bv:SUP(RN)) := [0,1]}
\texttt{H : Record( Hn:I, H1: SUP(I), H2: SUP(I)) := [0,1,x]}
\texttt{L : Record( Ln:I, L1: SUP(I), L2: SUP(I)) := [0,1,x]}
\texttt{P : Record( Pn:I, P1: SUP(RN), P2: SUP(RN)) := [0,1,y]}
\texttt{CT : Record( Tn:I, T1: SUP(I), T2: SUP(I)) := [0,1,x]}

U : Record( Un:I, U1:Sup(I), U2:Sup(I) ) := [0,1,0]

MonicQuotient: (Sup(I),Sup(I)) -> Sup(I)
MonicQuotient (a,b) ==
    leadingCoefficient(b) ^= 1 => error "divisor must be monic"
    b = 1 => a
    da := degree a
    db := degree b -- assertion: degree b > 0
    q:Sup(I) := 0
    while da >= db repeat
        t := monomial(leadingCoefficient a, (da-db)::NNI)
        a := a - b * t
        q := q + t
        da := degree a
    q

cyclotomic n ==
    --++ cyclotomic polynomial denoted \phi[n](x)
    p:I; q:I; r:I; s:I; m:NNI; c:Sup(I); t:Sup(I)
    n < 0 => error "cyclotomic not defined for negative integers"
    n = 0 => x
    k := n; s := p := 1
    c := x - 1
    while k > 1 repeat
        p := nextPrime p
        (q,r) := divide(k, p)
        if r = 0 then
            while r = 0 repeat (k := q; (q,r) := divide(k,p))
            t := multiplyExponents(c,p::NNI)
            c := MonicQuotient(t,c)
            s := s * p
        m := (n quo s) :: NNI
        multiplyExponents(c,m)

euler n ==
    p : Sup(RN); t : Sup(RN); c : RN; s : I
    n < 0 => error "euler not defined for negative integers"
    if n < E.En then (s,p) := (0$I,1$Sup(RN)) else (s,p) := E
    -- (s,p) := if n < E.En then (0,1) else E
    for i in s+1 .. n repeat
        t := (i::RN) * integrate p
        c := euler(i)$IntegerNumberTheoryFunctions / 2**(i::NNI) - t(1/2)
    p := t + c::Sup(RN)
    E.En := n
    E.Ev := p
    p

bernoulli n ==
    p : Sup(RN); t : Sup RN; c : RN; s : I
    n < 0 => error "bernoulli not defined for negative integers"
if \( n < B.Bn \) then \((s,p) := (0,I,1,SUP(RN))\) else \((s,p) := B\)

```dot
-- \((s,p) := if n < B.Bn then (0,1) else B\)
```

for \( i \) in \( s+1 .. n \) repeat
  \( t := (i::RN) \ast \text{integrate} \ p \)
  \( c := \text{bernoulli}(i)$\text{IntegerNumberTheoryFunctions}$\)
  \( p := t + c::SUP(RN)\)

\( B.Bn := n \)

\( B.Bv := p \)

```dot
\( P \)
```

\( \text{fixedDivisor a} == \)
  \( g:I; d:NNI; SUP(I)\)
  \( d := \text{degree a} \)
  \( g := \text{coefficient}(a, \text{minimumDegree a})\)
  for \( k \) in \( 1..d \) while \( g > 1 \) repeat \( g := \text{gcd}(g,a k)\)

\( g \)

\( \text{hermite n} == \)
  \( s : I; p : SUP(I); q : SUP(I)\)
  \( n < 0 \) => error "hermite not defined for negative integers"
  -- \((s,p,q) := if n < H.Hn then (0,1,x) else H\)
  if \( n < H.Hn \) then \((s := 0; p := 1; q := x)\) else \((s,p,q) := H\)
  for \( k \) in \( s+1 .. n \) repeat \((p,q) := (2*x*p-2*(k-1)*q,p)\)
  \( H.Hn := n \)
  \( H.H1 := p \)
  \( H.H2 := q \)

\( P \)

\( \text{legendre n} == \)
  \( s:I; t:I; p:SUP(RN); q:SUP(RN)\)
  \( n < 0 \) => error "legendre not defined for negative integers"
  -- \((s,p,q) := if n < P.Pn then (0,1,y) else P\)
  if \( n < P.Pn \) then \((s := 0; p := 1; q := y)\) else \((s,p,q) := P\)
  for \( k \) in \( s+1 .. n \) repeat
    \( t := k-1 \)
    \( (p,q) := ((k+t)$I/k*y*p - t/k*q,p)\)

\( P.Pn := n \)

\( P.P1 := p \)

\( P.P2 := q \)

\( P \)

\( \text{laguerre n} == \)
  \( k:I; s:I; t:I; p:SUP(I); q:SUP(I)\)
  \( n < 0 \) => error "laguerre not defined for negative integers"
  -- \((s,p,q) := if n < L.Ln then (0,1,x) else L\)
  if \( n < L.Ln \) then \((s := 0; p := 1; q := x)\) else \((s,p,q) := L\)
  for \( k \) in \( s+1 .. n \) repeat
    \( t := k-1 \)
    \( (p,q) := (((((k+t)$I)::SUP(I)-x)*p-t**2*q,p)\)

\( L.Ln := n \)
L.L1 := p
L.L2 := q
p

chebyshevT n ==
s : I; p : SUP(I); q : SUP(I)
n < 0 => error "chebyshevT not defined for negative integers"
-- (s,p,q) := if n < CT.Tn then (0,1,x) else CT
if n < CT.Tn then (s := 0; p := 1; q := x) else (s,p,q) := CT
for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
CT.Tn := n
CT.T1 := p
CT.T2 := q
p

chebyshevU n ==
s : I; p : SUP(I); q : SUP(I)
n < 0 => error "chebyshevU not defined for negative integers"
if n < U.Un then (s := 0; p := 1; q := 0) else (s,p,q) := U
for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
U.Un := n
U.U1 := p
U.U2 := q
p

package POLYROOT PolynomialRoots

— PolynomialRoots.input —

)set break resume
)sys rm -f PolynomialRoots.output
)spool PolynomialRoots.output
)set message test on
)set message auto off
CHAPTER 17. CHAPTER P

)clear all

--S 1 of 1
)show PolynomialRoots
--E 1

)spool
)lisp (bye)

— PolynomialRoots.help —

====================================================================
PolynomialRoots examples
====================================================================

Computes n-th roots of quotients of multivariate polynomials

See Also:
oc )show PolynomialRoots

——

PolynomialRoots (POLYROOT)

Exports:
froot nthr qroot rroot

—— package POLYROOT PolynomialRoots ——

)abbrev package POLYROOT PolynomialRoots
++ Author: Manuel Bronstein
++ Date Created: 15 July 1988
++ Date Last Updated: 10 November 1993
++ Description:
++ Computes n-th roots of quotients of multivariate polynomials
-- not visible to the user

**POLYROOT POLYNOMIALROOTS**

PolynomialRoots(E, V, R, P, F):Exports == Implementation where

E: OrderedAbelianMonoidSup
V: OrderedSet
R: IntegralDomain
P: PolynomialCategory(R, E, V)
F: Field with

numer : $ -> P
++ numer(x) undocumented
denom : $ -> P
++ denom(x) undocumented
coerce: P -> $
++ coerce(p) undocumented

N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
REC ==> Record(exponent:N, coef:F, radicand:F)

Exports ==> with

rroot: (R, N) -> REC
++ rroot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.
qroot : (Q, N) -> REC
++ qroot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.
if R has GcdDomain then froot: (F, N) -> REC
++ froot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.
nthr: (P, N) -> Record(exponent:N, coef:P, radicand:List P)
++ nthr(p,n) should be local but conditional

Implementation ==> add
import FactoredFunctions Z
import FactoredFunctions P
rsplit: List P -> Record(coef:R, poly:P)
zroot : (Z, N) -> Record(exponent:N, coef:Z, radicand:Z)

zroot(x, n) ==
-- zero? x or one? x => [1, x, 1]
zero? x or (x = 1) => [1, x, 1]
s := nthRoot(squareFree x, n)
[s.exponent, s.coef, */s.radicand]
if R has imaginary: () -> R then
  czroot: (Z, N) -> REC
  czroot(x, n) ==
    rec := zroot(x, n)
    rec.exponent = 2 and rec.radicand < 0 =>
      [rec.exponent, rec.coef * imaginary():P::F, (-rec.radicand)::F]
      [rec.exponent, rec.coef::F, rec.radicand::F]
  qroot(x, n) ==
    sn := czroot(numer x, n)
    sd := czroot(denom x, n)
    m := lcm(sn.exponent, sd.exponent)::N
    [m, sn.coef / sd.coef,
     (sn.radicand ** (m quo sn.exponent)) / (sd.radicand ** (m quo sd.exponent))]
    else
      qroot(x, n) ==
        sn := zroot(numer x, n)
        sd := zroot(denom x, n)
        m := lcm(sn.exponent, sd.exponent)::N
        [m, sn.coef::F / sd.coef::F,
         (sn.radicand ** (m quo sn.exponent))::F / (sd.radicand ** (m quo sd.exponent))::F]
  else
    if R has RetractableTo Fraction Z then
      rroot(x, n) ==
        (r := retractIfCan(x)@Union(Fraction Z, "failed")) case "failed"
        => [n, 1, x::P::F]
        qroot(r::Q, n)
    else
      rroot(x, n) ==
        (r := retractIfCan(x)@Union(Z, "failed")) case "failed"
        => [n, 1, x::P::F]
        qroot(r::Z::Q, n)
      else
        rroot(x, n) == [n, 1, x::P::F]
  else
    rsplit l ==
      r := 1$R
      p := 1$P
      for q in l repeat
        if (u := retractIfCan(q)@Union(R, "failed")) case "failed"
          then p := p * q
          else r := r * u::R
        [r, p]
if R has GcdDomain then
  if R has RetractableTo Z then
    nthr(x, n) ==
      (r := retractIfCan(x)@Union(Z,"failed")) case "failed"
      => nthRoot(squareFree x, n)
    rec := zroot(r::Z, n)
    [rec.exponent, rec.coef::P, [rec.radicand::P]]
  else nthr(x, n) == nthRoot(squareFree x, n)

froot(x, n) ==
  -- zero? x or one? x => [1, x, 1]
  zero? x or (x = 1) => [1, x, 1]
  sn := nthr(numer x, n)
  sd := nthr(denom x, n)
  pn := rsplit(sn.radicand)
  pd := rsplit(sd.radicand)
  rn := rroot(pn.coef, sn.exponent)
  rd := rroot(pd.coef, sd.exponent)
  m := lcm([rn.exponent, rd.exponent, sn.exponent, sd.exponent])::N
  [m, (sn.coef::F / sd.coef::F) * (rn.coef / rd.coef),
   ((rn.radicand ** (m quo rn.exponent)) / (rd.radicand ** (m quo rd.exponent))) *
   (pn.poly ** (m quo sn.exponent))::F /
   (pd.poly ** (m quo sd.exponent))::F]

package PSETPK PolynomialSetUtilitiesPackage

-- POLYROOT.dotabb --

"POLYROOT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYROOT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYROOT" -> "PFECAT"

package PSETPK PolynomialSetUtilitiesPackage

— PolynomialSetUtilitiesPackage.input —

)set break resume
)sys rm -f PolynomialSetUtilitiesPackage.output
)spool PolynomialSetUtilitiesPackage.output
)set message test on
This package provides modest routines for polynomial system solving. The aim of many of the operations of this package is to remove certain factors in some polynomials in order to avoid unnecessary computations in algorithms involving splitting techniques by partial factorization.

See Also:
- )show PolynomialSetUtilitiesPackage

Exports:
-- package PSETPK PolynomialSetUtilitiesPackage --

)abbrev package PSETPK PolynomialSetUtilitiesPackage
++ Author: Marc Moreno Maza (marc@mag.co.uk)
++ Date Created: 12/01/1995
++ Date Last Updated: 12/15/1998
++ Description:
++ This package provides modest routines for polynomial system solving.
++ The aim of many of the operations of this package is to remove certain
++ factors in some polynomials in order to avoid unnecessary computations
++ in algorithms involving splitting techniques by partial factorization.

PolynomialSetUtilitiesPackage (R,E,V,P) :Exports == Implementation where
removeRedundantFactors: LP -> LP
++ \texttt{removeRedundantFactors(lp)} returns \texttt{lq} such that if
++ \texttt{lp = \{p1,\ldots, pn\}} and \texttt{lq = \{q1, \ldots, qm\}}
++ then the product \texttt{p1*p2*\ldots*pn} vanishes iff the product \texttt{q1*q2*\ldots*qm} vanishes,
++ and the product of degrees of the \texttt{qi} is not greater than
++ the one of the \texttt{pj}, and no polynomial in \texttt{lq} divides another polynomial in \texttt{lq}.
++ In particular, ++ polynomials lying in the base ring \texttt{R} are removed.
++ Moreover, \texttt{lq} is sorted w.r.t \texttt{infRittWu?}.
++ Furthermore, if \texttt{R} is gcd-domain, the polynomials in \texttt{lq} are
++ pairwise without common non trivial factor.

removeRedundantFactors: (P,P) -> LP
++ \texttt{removeRedundantFactors(p,q)} returns the same as
++ \texttt{removeRedundantFactors([p,q])}

removeSquaresIfCan : LP -> LP
++ \texttt{removeSquaresIfCan(lp)} returns
++ \texttt{removeDuplicates [\texttt{squareFreePart(p)} \texttt{\texttt{P} \texttt{for } p \texttt{in lp}]} if \texttt{R} is gcd-domain else returns \texttt{lp}.

unprotectedRemoveRedundantFactors: (P,P) -> LP
++ \texttt{unprotectedRemoveRedundantFactors(p,q)} returns the same as
++ \texttt{removeRedundantFactors(p,q)} but does assume that neither
++ \texttt{p} nor \texttt{q} lie in the base ring \texttt{R} and assumes that
++ \texttt{infRittWu?(p,q)} holds. Moreover, if \texttt{R} is gcd-domain,
++ then \texttt{\texttt{axiom}[p]} and \texttt{\texttt{axiom}[q]} are assumed to be square free.

removeRedundantFactors: (LP,P) -> LP
++ \texttt{removeRedundantFactors(lp,q)} returns the same as
++ \texttt{removeRedundantFactors(cons(q,lp))} assuming
++ that \texttt{\texttt{removeRedundantFactors(lp)}} returns \texttt{lp}
++ up to replacing some polynomial \texttt{axiom[pj]} in \texttt{lpj}
++ by some some polynomial \texttt{axiom[qj]} associated to \texttt{axiom[pj]}

removeRedundantFactors : (LP,LP) -> LP
++ \texttt{removeRedundantFactors(lp,lq)} returns the same as
++ \texttt{removeRedundantFactors(concat(lp,lq))} assuming
++ that \texttt{\texttt{removeRedundantFactors(lp)}} returns \texttt{lp}
++ up to replacing some polynomial \texttt{axiom[pj]} in \texttt{lpj}
++ by some some polynomial \texttt{axiom[qj]} associated to \texttt{axiom[pj]}

removeRedundantFactors : (LP,LP,(LP -> LP)) -> LP
++ \texttt{removeRedundantFactors(lp,lq,remOp)} returns the same as
++ \texttt{concat(remOp(removeRoughlyRedundantFactorsInPols(lp,lq)),lq)}
++ assuming that \texttt{remOp(lq)} returns \texttt{lq} up to similarity.

certainlySubVariety? : (LP,LP) -> B
++ \texttt{certainlySubVariety?(newlp,lp)} returns true iff for every \texttt{axiom[p]}
++ \texttt{in \texttt{axiom[lp]}} the remainder of \texttt{axiom[p]} by \texttt{axiom[newlp]} using the division algorithm
++ of Groebner techniques is zero.

possiblyNewVariety? : (LP, List LP) -> B
++ \texttt{possiblyNewVariety?(newlp,lp)} returns true iff for every \texttt{axiom[lp]}
++ \texttt{in \texttt{axiom[llp]}} \texttt{certainlySubVariety?(newlp,lp)} does not hold.

probablyZeroDim?: LP -> B
++ \texttt{probablyZeroDim?(lp)} returns true iff the number of polynomials
++ in \texttt{axiom[lp]} is not smaller than the number of variables occurring
++ in these polynomials.
selectPolynomials : ((P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(selectPolynomials(pred?,ps))}\) returns \(\text{gps,bps}\) where
++ \(\text{axiom(gps)}\) is a list of the polynomial \(\text{axiom(p)}\) in \(\text{axiom(ps)}\)
++ such that \(\text{axiom(pred?(p))}\) holds and \(\text{axiom(bps)}\) are the other ones.
selectOrPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(selectOrPolynomials(lpred?,ps))}\) returns \(\text{gps,bps}\) where
++ \(\text{axiom(gps)}\) is a list of the polynomial \(\text{axiom(p)}\) in \(\text{axiom(ps)}\)
++ such that \(\text{axiom(pred?(p))}\) holds for some \(\text{axiom(pred?)}\) in \(\text{axiom(lpred?)}\)
++ and \(\text{axiom(bps)}\) are the other ones.
selectAndPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(selectAndPolynomials(lpred?,ps))}\) returns \(\text{gps,bps}\) where
++ \(\text{axiom(gps)}\) is a list of the polynomial \(\text{axiom(p)}\) in \(\text{axiom(ps)}\)
++ such that \(\text{axiom(pred?(p))}\) holds for every \(\text{axiom(pred?)}\) in \(\text{axiom(lpred?)}\)
++ and \(\text{axiom(bps)}\) are the other ones.
quasiMonicPolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(quasiMonicPolynomials(lp))}\) returns \(\text{qmps,nqmps}\) where
++ \(\text{axiom(qmps)}\) is a list of the quasi-monic polynomials in \(\text{axiom(lp)}\)
++ and \(\text{axiom(nqmps)}\) are the other ones.
univariate? : P -> B
++ \(\text{axiom(univariate?(p))}\) returns true iff \(\text{axiom(p)}\) involves one and
++ only one variable.
univariatePolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(univariatePolynomials(lp))}\) returns \(\text{ups,nups}\) where
++ \(\text{axiom(ups)}\) is a list of the univariate polynomials,
++ and \(\text{axiom(nups)}\) are the other ones.
linear? : P -> B
++ \(\text{axiom(linear?(p))}\) returns true iff \(\text{axiom(p)}\) does not lie
++ in the base ring \(\text{axiom(R)}\) and has main degree \(\text{axiom(1)}\).
linearPolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(linearPolynomials(lp))}\) returns \(\text{lps,nlps}\) where
++ \(\text{axiom(lps)}\) is a list of the linear polynomials in \(\text{axiom(lp)}\)
++ and \(\text{axiom(nlps)}\) are the other ones.
bivariate? : P -> B
++ \(\text{axiom(bivariate?(p))}\) returns true iff \(\text{axiom(p)}\) involves two and
++ only two variables.
bivariatePolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \(\text{axiom(bivariatePolynomials(lp))}\) returns \(\text{bps,nbps}\) where
++ \(\text{axiom(bps)}\) is a list of the bivariate polynomials,
++ and \(\text{axiom(nbps)}\) are the other ones.
removeRoughlyRedundantFactorsInPols : (LP, LP) -> LP
++ \(\text{axiom(removeRoughlyRedundantFactorsInPols(lp,lp1))}\) returns
++ \(\text{axiom(newlp)}\) where \(\text{axiom(newlp)}\) is obtained from \(\text{axiom(lp1)}\)
++ by removing in every polynomial \(\text{axiom(p)}\) of \(\text{axiom(lp)}\)
++ any occurrence of a polynomial \(\text{axiom(f)}\) in \(\text{axiom(lp1)}\).
++ This may involve a lot of exact-quotients computations.
removeRoughlyRedundantFactorsInPols : (LP, LP,B) -> LP
++ \(\text{axiom(removeRoughlyRedundantFactorsInPols(lp,lp1,opt))}\) returns
++ the same as \(\text{axiom(removeRoughlyRedundantFactorsInPols(lp,lp1))}\)
++ if \(\text{axiom(opt)}\) is \(\text{axiom(false)}\) and if the previous operation
++ does not return any non null and constant polynomial, 
++ else return \texttt{\{1\}}.

\texttt{removeRoughlyRedundantFactorsInPol : (P,LP) -> P}
++ \texttt{removeRoughlyRedundantFactorsInPol(p,lf)} returns the same as
++ \texttt{removeRoughlyRedundantFactorsInPols([p],lf,true)}

\texttt{interReduce: LP -> LP}
++ \texttt{interReduce(lp)} returns \texttt{lp} such that \texttt{lp} and \texttt{lq}
++ in \texttt{lp} is reducible by the others in the sense
++ of Groebner bases. Since no assumptions are required
++ the result may depend on the ordering the reductions are
++ performed.

\texttt{roughBasicSet: LP -> Union(Record(bas:T,top:LP),"failed")}
++ \texttt{roughBasicSet(lp)} returns the smallest (with Ritt-Wu
++ ordering) triangular set contained in \texttt{lp}.

\texttt{crushedSet: LP -> LP}
++ \texttt{crushedSet(lp)} returns \texttt{lq} such that \texttt{lp} and
++ \texttt{lq} generate the same ideal and no rough basic
++ sets reduce (in the sense of Groebner bases) the other
++ polynomials in \texttt{lq}.

\texttt{rewriteSetByReducingWithParticularGenerators : (LP,(P->B),((P,P)->B),((P,P)->P)) -> LP}
++ \texttt{rewriteSetByReducingWithParticularGenerators(lp,pred?,redOp?,redOp)}
++ returns \texttt{lq} where \texttt{lq} is computed by the following
++ algorithm. Chose a basic set w.r.t. the reduction-test \texttt{redOp?}
++ among the polynomials satisfying property \texttt{redOp?},
++ if it is empty then leave, else reduce the other polynomials by
++ this basic set w.r.t. the reduction-operation \texttt{redOp}.
++ Repeat while another basic set with smaller rank can be computed.
++ See code. If \texttt{pred?} is \texttt{quasiMonic?} the ideal is unchanged.

\texttt{rewriteIdealWithQuasiMonicGenerators : (LP,((P,P)->B),((P,P)->P)) -> LP}
++ \texttt{rewriteIdealWithQuasiMonicGenerators(lp,redOp?,redOp)} returns
++ \texttt{lq} where \texttt{lq} is computed by reducing \texttt{lp}
++ w.r.t. some basic set of the ideal generated by
++ the quasi-monic polynomials in \texttt{lp}.

\texttt{if R has GcdDomain}
then

\texttt{squareFreeFactors : P -> LP}
++ \texttt{squareFreeFactors(p)} returns the square-free factors of \texttt{p}
++ over \texttt{R}.

\texttt{univariatePolynomialsGcds : LP -> LP}
++ \texttt{univariatePolynomialsGcds(lp)} returns \texttt{lq} where
++ \texttt{lq} is a list of the gcds of every pair in \texttt{lp}
++ of univariate polynomials in the same main variable.

\texttt{univariatePolynomialsGcds : (LP,B) -> LP}
++ \texttt{univariatePolynomialsGcds(lp,opt)} returns the same as
++ \texttt{univariatePolynomialsGcds(lp)} if \texttt{opt} is
++ \texttt{false} and if the previous operation does not return
++ any non null and constant polynomial, else return \axiom{[1]}.
removeRoughlyRedundantFactorsInContents : (LP, LP) -> LP
++ \axiom{removeRoughlyRedundantFactorsInContents(lp,lf)} returns
++ \axiom{newlp} where \axiom{newlp} is obtained from \axiom{lp}
++ by removing in the content of every polynomial of \axiom{lp}
++ any occurrence of a polynomial \axiom{f} in \axiom{lf}. Moreover,
++ squares over \axiom{R} are first removed in the content
++ of every polynomial of \axiom{lp}.
removeRedundantFactorsInContents : (LP, LP) -> LP
++ \axiom{removeRedundantFactorsInContents(lp,lf)} returns \axiom{newlp}
++ where \axiom{newlp} is obtained from \axiom{lp} by removing
++ in the content of every polynomial of \axiom{lp} any non trivial
++ factor of any polynomial \axiom{f} in \axiom{lf}. Moreover,
++ squares over \axiom{R} are first removed in the content
++ of every polynomial of \axiom{lp}.
removeRedundantFactorsInPols : (LP, LP) -> LP
++ \axiom{removeRedundantFactorsInPols(lp,lf)} returns \axiom{newlp}
++ where \axiom{newlp} is obtained from \axiom{lp} by removing
++ in every polynomial \axiom{p} of \axiom{lp} any non trivial
++ factor of any polynomial \axiom{f} in \axiom{lf}. Moreover,
++ squares over \axiom{R} are first removed in every
++ polynomial \axiom{lp}.
if (R has EuclideanDomain) and (R has CharacteristicZero)
then

irreducibleFactors : LP -> LP
++ \axiom{irreducibleFactors(lp)} returns \axiom{lf} such that if
++ \axiom{lp = [p1,...,pn]} and \axiom{lf = [f1,...,fm]} then
++ \axiom{p1*p2*...*pn=0} means \axiom{f1*f2*...*fm=0}, and the \axiom{fi}
++ are irreducible over \axiom{R} and are pairwise distinct.
lazyIrreducibleFactors : LP -> LP
++ \axiom{lazyIrreducibleFactors(lp)} returns \axiom{lf} such that if
++ \axiom{lp = [p1,...,pn]} and \axiom{lf = [f1,...,fm]} then
++ \axiom{p1*p2*...*pn=0} means \axiom{f1*f2*...*fm=0}, and the \axiom{fi}
++ are irreducible over \axiom{R} and are pairwise distinct.
++ The algorithm tries to avoid factorization into irreducible
++ factors as far as possible and makes previously use of gcd
++ techniques over \axiom{R}.
removeIrreducibleRedundantFactors : (LP, LP) -> LP
++ \axiom{removeIrreducibleRedundantFactors(lp,lq)} returns the same
++ as \axiom{irreducibleFactors(concat(lp,lq))} assuming
++ that \axiom{irreducibleFactors(lp)} returns \axiom{lp}
++ up to replacing some polynomial \axiom{pj} in \axiom{lp}
++ by some polynomial \axiom{qj} associated to \axiom{pj}.

Implementation == add

autoRemainder: T -> List(P)
removeAssociates (lp:LP):LP ==
removeDuplicates [primPartElseUnitCanonical(p) for p in lp]
selectPolynomials (pred?, ps) ==
gps : LP := []
bps : LP := []
while not empty? ps repeat
    p := first ps
    ps := rest ps
    if pred?(p)
        then
            gps := cons(p,gps)
        else
            bps := cons(p,bps)
gps := sort(infRittWu?,gps)
bps := sort(infRittWu?,bps)
[gps,bps]

selectOrPolynomials (lpred?, ps) ==
gps : LP := []
bps : LP := []
while not empty? ps repeat
    p := first ps
    ps := rest ps
    clpred? := lpred?
    while (not empty? clpred?) and (not (first clpred?)(p)) repeat
        clpred? := rest clpred?
    if not empty?(clpred?)
        then
            gps := cons(p,gps)
        else
            bps := cons(p,bps)
gps := sort(infRittWu?,gps)
bps := sort(infRittWu?,bps)
[gps,bps]

selectAndPolynomials (lpred?, ps) ==
gps : LP := []
bps : LP := []
while not empty? ps repeat
    p := first ps
    ps := rest ps
    clpred? := lpred?
    while (not empty? clpred?) and ((first clpred?)(p)) repeat
        clpred? := rest clpred?
    if empty?(clpred?)
        then
            gps := cons(p,gps)
        else
            bps := cons(p,bps)
gps := sort(infRittWu?,gps)
bps := sort(infRittWu?,bps)
[gps,bps]

linear? p ==
  ground? p => false
-- one?(mdeg(p))
  (mdeg(p) == 1)

linearPolynomials ps ==
  selectPolynomials(linear?,ps)

univariate? p ==
  ground? p => false
  not(ground?(init(p))) => false
  tp := tail(p)
  ground?(tp) => true
  not (mvar(p) = mvar(tp)) => false
  univariate?(tp)

univariatePolynomials ps ==
  selectPolynomials(univariate?,ps)

bivariate? p ==
  ground? p => false
  ground? tail(p) => univariate?(init(p))
  vp := mvar(p)
  vtp := mvar(tail(p))
  ((ground?(init(p)) and (vp = vtp)) => bivariate? tail(p)
  ((ground?(init(p)) and (vp > vtp)) => univariate? tail(p)
  not univariate?(init(p)) => false
  vip := mvar(init(p))
  vip > vtp => false
  vip = vtp => univariate? tail(p)
  vtp < vp => false
  zero? degree(tail(p),vip) => univariate? tail(p)
  bivariate? tail(p)

bivariatePolynomials ps ==
  selectPolynomials(bivariate?,ps)

quasiMonicPolynomials ps ==
  selectPolynomials(quasiMonic?,ps)

removeRoughlyRedundantFactorsInPols (lp,lf,opt) ==
  empty? lp => lp
  newlp : LP := []
  stop : B := false
  lp := remove(zero?,lp)
  lf := sort(infRittWu?,lf)
  test : Union(P,"failed")
  while (not empty? lp) and (not stop) repeat
p := first lp
lp := rest lp
copylf := lf
while (not empty? copylf) and (not ground? p) and (not (mvar(p) < mvar(first copylf)))
  f := first copylf
  copylf := rest copylf
  while (((test := p exquo$P f)) case P) repeat
    p := test::P
  stop := opt and ground?(p)
  newlp := cons(unitCanonical(p),newlp)
stop => [1$P]
newlp

removeRoughlyRedundantFactorsInPol(p,lf) ==
  zero? p => p
  lp : LP := [p]
  first removeRoughlyRedundantFactorsInPol(lp,lf,unitCanonical)(lp,lf,unitCanonical)

removeRoughlyRedundantFactorsInPol(lp,lf) ==
  removeRoughlyRedundantFactorsInPol(lp,lf,unitCanonical)

possiblyNewVariety?(newlp,llp) ==
  while (not empty? llp) and _
    (not certainlySubVariety?(newlp,first(llp))) repeat
    llp := rest llp
  empty? llp

certainlySubVariety?(lp,lq) ==
  gs := construct(lp)$GPS
  while (not empty? lq) and _
    (zero? (remainder(first(lq),gs)$GPS).polnum) repeat
    lq := rest lq
  empty? lq

probablyZeroDim?(lp: List P) : Boolean ==
  m := #lp
  lv : List V := variables(first lp)
  while not empty? (lp := rest lp) repeat
    lv := concat(variables(first lp),lv)
  n := #(removeDuplicates lv)
  not (n > m)

interReduce(lp: LP): LP ==
  ps := lp
r : List(P) := []
  repeat
    empty? ps => return rs
    ps := sort(supRittWu?, ps)
  p := first ps
  ps := rest ps
\[
\begin{align*}
\text{roughRed?(p:P,q:P):B} &= \text{ground? p} \Rightarrow \text{false} \\
&\quad \text{ground? q} \Rightarrow \text{true} \\
&\quad \text{mvar}(p) > \text{mvar}(q)
\end{align*}
\]

\[
\text{roughBasicSet(lp) == basicSet(lp, roughRed?)$T}
\]

\[
\text{autoRemainder(ts:T): List(P) ==}
\]
\[
\begin{align*}
\text{empty? ts} &\Rightarrow \text{members(ts)} \\
\text{lp} &:= \text{sort(infRittWu?, reverse members(ts))} \\
\text{newlp : List(P) := [primPartElseUnitCanonical first(lp)]} \\
\text{lp} &:= \text{rest(lp)} \\
\text{while not empty? lp repeat} \\
\text{p} &:= (\text{remainder(first(lp), construct(newlp)$GPS)$GPS).polnum} \\
\text{if not zero? p} &\text{ then} \\
\text{if ground? p} &\text{ then} \\
\text{newlp} &:= [1$P] \\
\text{lp} &:= [] \\
\text{else} \\
\text{newlp} &:= \text{cons(p, newlp)} \\
\text{lp} &:= \text{rest(lp)} \\
\text{else} \\
\text{lp} &:= \text{rest(lp)} \\
\text{newlp}
\end{align*}
\]

\[
\text{crushedSet(lp) ==}
\]
\[
\begin{align*}
\text{rec := roughBasicSet(lp)} \\
\text{contradiction := (rec case "failed")@B} \\
\text{finished : B := false} \\
\text{while (not finished) and (not contradiction) repeat} \\
\text{bs := (rec::RBT).bas} \\
\text{rs := (rec::RBT).top} \\
\text{rs := rewriteIdealWithRemainder(rs, bs)$T} \\
\text{-- contradiction := ((not empty? rs) and (one? first(rs)))} \\
\text{contradiction := ((not empty? rs) and (first(rs) = 1))} \\
\text{if not contradiction} &\text{ then} \\
\text{rs} &:= \text{concat(rs, autoRemainder(bs))} \\
\text{rec} &:= \text{roughBasicSet(rs)} \\
\text{contradiction} &:= (\text{rec case "failed")@B} \\
\text{not contradiction} &\Rightarrow \text{finished := not infRittWu?((rec::RBT).bas, bs)}
\end{align*}
\]
CHAPTER 17. CHAPTER P

contradiction => [1$P]
rs

rewriteSetByReducingWithParticularGenerators (ps,pred?,redOp?,redOp) ==
rs : LP := remove(zero?,ps)
any?(ground?,rs) => [1$P]
contradiction : B := false
bs1 : T := empty()$T
rec : Union(RBT,"failed")
ar : Union(T,List(P))
stop : B := false
while (not contradiction) and (not stop) repeat
rec := basicSet(rs,pred?,redOp?)$T
bs2 : T := (rec::RBT).bas
rs := (rec::RBT).top
-- ar := autoReduce(bs2,lazyPrem,reduced?)@Union(T,List(P))
ar := bs2::Union(T,List(P))
if (ar case T)@B
then
bs2 := ar::T
if infrittwu?(bs2,bs1)
then
rs := rewriteSetWithReduction(rs,bs2,redOp,redOp?)$T
bs1 := bs2
else
stop := true
rs := concat(members(bs2),rs)
else
rs := concat(ar::LP,rs)
if any?(ground?,rs)
then
contradiction := true
rs := [1$P]
rs

removeRedundantFactors (lp:LP,lq :LP, remOp : (LP -> LP)) ==
-- ASSUME remOp(lp) returns lp up to similarity
lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
lq := remOp lq
sort(infrittwu?,concat(lp,lq))

removeRedundantFactors (lp:LP,lq :LP) ==
lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
lq := removeRedundantFactors lq
sort(infrittwu?,concat(lp,lq))

if (R has EuclideanDomain) and (R has CharacteristicZero)
then
irreducibleFactors lp ==
newlp : LP := []
lrrz : List RRZ
rrz : RRZ
fp : FP
while not empty? lp repeat
  p := first lp
  lp := rest lp
  fp := factor(p)$pf
  lrrz := factors(fp)$FP
  lf := remove(ground?, [rrz.factor for rrz in lrrz])
  newlp := concat(lf, newlp)
removeDuplicates newlp

lazyIrreducibleFactors lp ==
  lp := removeRedundantFactors(lp)
  newlp : LP := []
  lrrz : List RRZ
  rrz : RRZ
  fp : FP
  while not empty? lp repeat
    p := first lp
    lp := rest lp
    fp := factor(p)$pf
    lrrz := factors(fp)$FP
    lf := remove(ground?, [rrz.factor for rrz in lrrz])
    newlp := concat(lf, newlp)
  newlp

removeIrreducibleRedundantFactors (lp:LP, lq :LP) ==
  -- ASSUME lp only contains irreducible factors over R
  lq := removeRoughlyRedundantFactorsInPols(lq, lp, false)
  lq := irreducibleFactors lq
  sort(infRittWu?, concat(lp, lq))

if R has GcdDomain
  then

  squareFreeFactors(p:P) ==
    sfp: Factored P := squareFree(p)$P
    lsf: List P := [foo.factor for foo in factors(sfp)]
    lsf

  univariatePolynomialsGcds (ps, opt) ==
    lg : LP := []
    pInV : LP
    stop : B := false
    ps := sort(infRittWu?, ps)
    p, g : P
    v : V
    while (not empty? ps) and (not stop) repeat
      while (not empty? ps) and (not univariate?((p := first(ps)))) repeat
univariatePolynomialsGcds ps ==
  univariatePolynomialsGcds (ps,false)

removeSquaresIfCan lp ==
  empty? lp => lp
  removeDuplicates [squareFreePart(p)$P for p in lp]

rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp?) ==
  ups := removeSquaresIfCan(univariatePolynomialsGcds(ps,true))
  ps := removeDuplicates concat(ups,ps)
  rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp?)

removeRoughlyRedundantFactorsInContents (ps,lf) ==
  empty? ps => ps
  newps : LP := []
  p,newp,cp,newcp,f,g : P
  test : Union(P,"failed")
  copylf : LP
  while not empty? ps repeat
    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newcp
      then
        copylf := [f for f in lf | mvar(f) <= mvar(newcp)]
        while (not empty? copylf) and (not ground? newcp) repeat
          f := first copylf
          copylf := rest copylf
          test := (newcp exquo$P f)
          if (test case P)@B
            then
              newcp := test::P
    else
      newps := cons(p,newps)

    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newcp
      then
        copylf := [f for f in lf | mvar(f) <= mvar(newcp)]
        while (not empty? copylf) and (not ground? newcp) repeat
          f := first copylf
          copylf := rest copylf
          test := (newcp exquo$P f)
          if (test case P)@B
            then
              newcp := test::P
if ground? newcp 
    then 
        newp := unitCanonical(newp)
    else 
        newp := unitCanonical(newp * newcp)
    newps := cons(newp,newps)
    newps

removeRedundantFactorsInContents (ps,lf) ==
    empty? ps => ps
    newps : LP := []
    p,newp,cp,newcp,f,g : P
    while not empty? ps repeat
        p := first ps
        ps := rest ps
        cp := mainContent(p)$P
        newcp := squareFreePart(cp)$P
        newp := (p exquo$P cp)::P
        if not ground? newcp 
            then 
                copylf := lf
                while (not empty? copylf) and (not ground? newcp) repeat
                    f := first copylf
                    copylf := rest copylf
                    g := gcd(newcp,f)$P
                    if not ground? g 
                        then 
                            newcp := (newcp exquo$P g)::P
                    if ground? newcp 
                        then 
                            newp := unitCanonical(newp)
                        else 
                            newp := unitCanonical(newp * newcp)
                newps := cons(newp,newps)
    newps

removeRedundantFactorsInPols (ps,lf) ==
    empty? ps => ps
    newps : LP := []
    p,newp,cp,newcp,f,g : P
    while not empty? ps repeat
        p := first ps
        ps := rest ps
        cp := mainContent(p)$P
        newcp := squareFreePart(cp)$P
        newp := (p exquo$P cp)::P
        newp := squareFreePart(newp)$P
        copylf := lf
        while not empty? copylf repeat
            f := first copylf 
            copylf := rest copylf
            if ground? newcp 
                then 
                    newp := unitCanonical(newp)
                else 
                    newp := unitCanonical(newp * newcp)
            newps := cons(newp,newps)
    newps
copylf := rest copylf
if not ground? newcp
  then
    g := gcd(newcp,f)$P
    if not ground? g
      then
        newcp := (newcp exquo$P g)::P
    if not ground? newp
      then
        g := gcd(newp,f)$P
        if not ground? g
          then
            newp := (newp exquo$P g)::P
    if ground? newcp
      then
        newp := unitCanonical(newp)
    else
      newp := unitCanonical(newp * newcp)
newps := cons(newp,newps)
newps

removeRedundantFactors (a:P,b:P) : LP ==
a := primPartElseUnitCanonical(squareFreePart(a))
b := primPartElseUnitCanonical(squareFreePart(b))
if not infRittWu?(a,b)
  then
    (a,b) := (b,a)
if ground? a
  then
    if ground? b
      then
        return([])
      else
        return([b])
  else
    if ground? b
      then
        return([a])
    else
      return(unprotectedRemoveRedundantFactors(a,b))

unprotectedRemoveRedundantFactors (a,b) ==
c := b exquo$P a
if (c case P)$P
  then
    d : P := c::P
    if ground? d
      then
        return([a])
    else
      return([b])
else
  return([a,b])
return([a,d])
else
g : P := gcd(a,b)$P
if ground? g
then
    return([a,b])
else
    return([g,(a exquo$P g)::P,(b exquo$P g)::P])
else

removeSquaresIfCan lp ==
    lp
rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp?) ==
    rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp?)

removeRedundantFactors (a:P,b:P) ==
a := primPartElseUnitCanonical(a)
b := primPartElseUnitCanonical(b)
if not infRittWu?(a,b)
    then
        (a,b) := (b,a)
    if ground? a
        then
            if ground? b
                then
                    return([])
                else
                    return([b])
        else
            return([a])
    else
        return(unprotectedRemoveRedundantFactors(a,b))

unprotectedRemoveRedundantFactors (a,b) ==
c := b exquo$P a
if (c case P)$B
    then
d : P := c::P
    if ground? d
        then
            return([a])
        else
            if infRittWu?(d,a) then (a,d) := (d,a)
                return(unprotectedRemoveRedundantFactors(a,d))
            else
                return([a,b])
removeRedundantFactors (lp:LP) ==
lp := remove(ground?, lp)
lp := removeDuplicates [primPartElseUnitCanonical(p) for p in lp]
lp := removeSquaresIfCan lp
lp := removeDuplicates [unitCanonical(p) for p in lp]
empty? lp => lp
size?(lp,1$N)(List P) => lp
lp := sort(infRittWu?,lp)
p : P := first lp
lp := rest lp
base : LP := unprotectedRemoveRedundantFactors(p,first lp)
top : LP := rest lp
while not empty? top repeat
  p := first top
  base := removeRedundantFactors(base,p)
top := rest top
base

removeRedundantFactors (lp:LP,a:P) ==
lp := remove(ground?, lp)
lp := sort(infRittWu?, lp)
ground? a => lp
empty? lp => [a]
toSee : LP := lp
toSave : LP := []
while not empty? toSee repeat
  b := first toSee
toSee := rest toSee
if not infRittWu?(b,a)
  then
    (c,d) := (a,b)
  else
    (c,d) := (b,a)
rrf := unprotectedRemoveRedundantFactors(c,d)
empty? rrf => error"in removeRedundantFactors : (LP,P) -> LP from PSETPK"
c := first rrf
rrf := rest rrf
if empty? rrf
  then
    if associates?(c,b)
      then
        toSave := concat(toSave,toSee)
        a := b
toSee := []
    else
      a := c
toSee := concat(toSave,toSee)
toSave := []
  else

d := first rrf
rrf := rest rrf
if empty? rrf
  then
    if associates?(c,b)
      then
toSave := concat(toSave,[b])
a := d
    else
      if associates?(d,b)
        then
toSave := concat(toSave,[b])
a := c
      else
toSave := removeRedundantFactors(toSave,c)
a := d
    else
e := first rrf
not empty? rest(rrf) => error"in removeRedundantFactors:(LP,P)->LP from PSETPK"
-- ASSUME that neither c, nor d, nor e may be associated to b
toSave := removeRedundantFactors(toSave,c)
toSave := removeRedundantFactors(toSave,d)
a := e
if empty? toSee
  then
toSave := sort(infRittWu?,cons(a,toSave))
toSave

—— PSETPK.dotabb ——
"PSETPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSETPK"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"PSETPK" -> "RPOLCAT"

—— package SOLVEFOR PolynomialSolveByFormulas ——
"PSETPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSETPK"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"PSETPK" -> "RPOLCAT"

—— PolynomialSolveByFormulas.input ——
)set break resume
)sys rm -f PolynomialSolveByFormulas.output
PolynomialSolveByFormulas (SOLVEFOR)

Exports:
aCubic aLinear aQuadratic aQuartic cubic linear particularSolution quadratic quartic solve mapSolve

— package SOLVEFOR PolynomialSolveByFormulas —

)abbrev package SOLVEFOR PolynomialSolveByFormulas
++ Author: SMW June 86, BMT Sept 93
++ Description:
++ This package factors the formulas out of the general solve code,
++ allowing their recursive use over different domains.
++ Care is taken to introduce few radicals so that radical extension
++ domains can more easily simplify the results.

PolynomialSolveByFormulas(UP, F): PSFcat == PSFdef where

UP: UnivariatePolynomialCategory F
F: Field with "**": (% F, Fraction Integer) -> %

L ==> List
PSFcat == with
  solve: UP -> L F
  ++ solve(u) \undocumented
  particularSolution: UP -> F
  ++ particularSolution(u) \undocumented
  mapSolve: (UP, F -> F) -> Record(solns: L F,
    maps: L Record(arg:F,res:F))
  ++ mapSolve(u,f) \undocumented

linear: UNIVAR -> L F
  ++ linear(u) \undocumented
quadratic: UP -> L F
  ++ quadratic(u) \undocumented
cubic: UP -> L F
  ++ cubic(u) \undocumented
quartic: UP -> L F
  ++ quartic(u) \undocumented

-- Arguments give coefs from high to low degree.
linear: (F, F) -> L F
  ++ linear(f,g) \undocumented
quadratic: (F, F, F) -> L F
  ++ quadratic(f,g,h) \undocumented
cubic: (F, F, F, F) -> L F
  ++ cubic(f,g,h,i) \undocumented
quartic: (F, F, F, F, F) -> L F
  ++ quartic(f,g,h,i,j) \undocumented

aLinear: (F, F) -> F
++ aLinear(f,g) \undocumented
aQuadratic: (F, F, F) -> F  
++ aQuadratic(f,g,h) \undocumented
aCubic: (F, F, F, F) -> F  
++ aCubic(f,g,h,j) \undocumented
aQuartic: (F, F, F, F, F) -> F  
++ aQuartic(f,g,h,i,k) \undocumented

PSFdef == add
-----------------------------------------------------------------
-- Stuff for mapSolve
-----------------------------------------------------------------

id ==> (IDENTITY$Lisp)

maplist: List Record(arg: F, res: F) := []
mapSolving?: Boolean := false
-- map: F -> F := id #1 replaced with line below
map: Boolean := false

mapSolve(p, fn) ==
-- map := fn #1 replaced with line below
locmap: F -> F := x +-> fn x; map := id locmap
mapSolving? := true; maplist := []
alist := solve p
mapSolving? := false;
-- map := id #1 replaced with line below
locmap := x +-> id x; map := id locmap
[slist, maplist]

part(s: F): F ==
not mapSolving? => s
-- t := map s replaced with line below
t: F := SPADCALL(s, map)$Lisp
t = s => s
maplist := cons([t, s], maplist)
t

-----------------------------------------------------------------
-- Entry points and error handling
-----------------------------------------------------------------

cr ==> coefficient

-- local intsolve
intsolve(u:UP):L(F) ==
u := (factors squareFree u).1.factor
n := degree u
n=1 => linear (cc(u,1), cc(u,0))
n=2 => quadratic (cc(u,2), cc(u,1), cc(u,0))
n=3 => cubic (cc(u,3), cc(u,2), cc(u,1), cc(u,0))
n=4 => quartic \((cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))\)
error "All sqfr factors of polynomial must be of degree < 5"

solve u ==
ls := nil$L(F)
for f in factors squareFree u repeat
  lsf := intsolve f.factor
  for i in 1..(f.exponent) repeat ls := [:lsf,:ls]
  ls

particularSolution u ==
  u := (factors squareFree u).1.factor
  n := degree u
  n=1 => aLinear \((cc(u,1), cc(u,0))\)
  n=2 => aQuadratic \((cc(u,2), cc(u,1), cc(u,0))\)
  n=3 => aCubic \((cc(u,3), cc(u,2), cc(u,1), cc(u,0))\)
  n=4 => aQuartic \((cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))\)
  error "All sqfr factors of polynomial must be of degree < 5"

needDegree(n: Integer, u: UP): Boolean ==
degree u = n => true
error concat("Polynomial must be of degree ", n::String)

needLcoef(cn: F): Boolean ==
cn ^= 0 => true
error "Leading coefficient must not be 0."

needChar0(): Boolean ==
characteristic()$F = 0 => true
error "Formula defined only for fields of characteristic 0."

linear u ==
  needDegree(1, u)
  linear \((coefficient(u,1), coefficient(u,0))\)

quadratic u ==
  needDegree(2, u)
  quadratic \((coefficient(u,2), coefficient(u,1), coefficient(u,0))\)

cubic u ==
  needDegree(3, u)
  cubic \((coefficient(u,3), coefficient(u,2), coefficient(u,1), coefficient(u,0))\)

quartic u ==
  needDegree(4, u)
  quartic \((coefficient(u,4), coefficient(u,3), coefficient(u,2), coefficient(u,1), coefficient(u,0))\)
-- The formulas
-- local function for testing equality of radicals.
-- This function is necessary to detect at least some of the
-- situations like sqrt(9)-3 = 0 --> false.
equ(x:F,y:F):Boolean ==
  ( (recip(x-y)) case "failed" ) => true
  false

linear(c1, c0) ==
  needLcoef c1
  [- c0/c1 ]

aLinear(c1, c0) ==
  first linear(c1,c0)

quadratic(c2, c1, c0) ==
  needLcoef c2; needChar0()
  (c0 = 0) => [0$F,:linear(c2, c1)]
  (c1 = 0) => [(-c0/c2)**(1/2),(-c0/c2)**(1/2)]
  D := part(c1**2 - 4*c2*c0)**(1/2)
  [(-c1+D)/(2*c2), (-c1-D)/(2*c2)]

aQuadratic(c2, c1, c0) ==
  needLcoef c2; needChar0()
  (c0 = 0) => 0$F
  (c1 = 0) => (-c0/c2)**(1/2)
  D := part(c1**2 - 4*c2*c0)**(1/2)
  (-c1+D)/(2*c2)

w3: F := (-1 + (-3::F)**(1/2)) / 2::F

cubic(c3, c2, c1, c0) ==
  needLcoef c3; needChar0()
  -- case one root = 0, not necessary but keeps result small
  (c0 = 0) => [0$F,:quadratic(c3, c2, c1)]
  a1 := c2/c3; a2 := c1/c3; a3 := c0/c3

  -- case x**3-a3 = 0, not necessary but keeps result small
  (a1 = 0 and a2 = 0) =>
    [ u*(-a3)**(1/3) for u in [1, w3, w3**2 ] ]

  -- case x**3 + a1*x**2 + a1**2*x/3 + a3 = 0, the general for-
  -- mula is not valid in this case, but solution is easy.
P := part(-a1/3::F)
equ(a1**2,3*a2) =>
  S := part((- a3 + (a1**3)/(27::F)**(1/3)))
\[
\text{[ P + S*u for u in [1, w3, w3**2] ]}
\]

-- general case
\[
Q := \text{part}\left(\left(3*a2 - a1**2\right)/9::F\right)
\]
\[
R := \text{part}\left(\left(9*a1*a2 - 27*a3 - 2*a1**3\right)/54::F\right)
\]
\[
D := \text{part}\left(Q**3 + R**2\right)**(1/2)
\]
\[
S := \text{part}\left(R + D\right)**(1/3)
\]
-- S = 0 is done in the previous case
\[
\text{[ P + S*u - Q/(S*u) for u in [1, w3, w3**2] ]}
\]

\begin{align*}
aCubic(c3, c2, c1, c0) &= \text{needLcoef c3; needChar0()}
\text{(c0 = 0) } & \Rightarrow 0$F
a1 := c2/c3; a2 := c1/c3; a3 := c0/c3
(a1 = 0 and a2 = 0) & \Rightarrow (-a3)**(1/3)
P := \text{part}\left(-a1/3::F\right)
equ(a1**2,3*a2) &=
S := \text{part}\left(- a3 + (a1**3)/27::F\right)**(1/3)
P + S
Q := \text{part}\left(\left(3*a2 - a1**2\right)/9::F\right)
R := \text{part}\left(\left(9*a1*a2 - 27*a3 - 2*a1**3\right)/54::F\right)
D := \text{part}\left(Q**3 + R**2\right)**(1/2)
S := \text{part}\left(R + D\right)**(1/3)
P + S - Q/S
\end{align*}

\begin{align*}
quartic(c4, c3, c2, c1, c0) &= \text{needLcoef c4; needChar0()}
\text{(c0 = 0) } & \Rightarrow [0$F,:cubic(c4, c3, c2, c1)]
\text{Make monic:}
a1 := c3/c4; a2 := c2/c4; a3 := c1/c4; a4 := c0/c4
\text{(a1 = 0 and a2 = 0 and a3 = 0) } & \Rightarrow
\text{append( quadratic(1, 0, (-a4)**(1/2)), _}
\text{quadratic(1 ,0, -((-a4)**(1/2))) )}
\end{align*}

-- Translate w = x+a1/4 to eliminate a1:  w**4+p*w**2+q*w+r
p := \text{part}\left(a2-3*a1*a1/8::F\right)
q := \text{part}\left(a3-a1*a2/2::F + a1**3/8::F\right)
r := \text{part}\left(a4-a1*a3/4::F + a1**2*a2/16::F - 3*a1**4/256::F\right)
t0 := the cubic resolvent of x**3-p*x**2-4*r*x**2+4*p*r-q**2
-- The roots of the translated polynomial are those of
-- two quadratics. (What about rt=0 ?)
-- rt=0 can be avoided by picking a root `^= p of the cubic
-- polynomial above. This is always possible provided that
-- the input is squarefree. In this case the two other roots
-- are +(-) 2*r**(1/2).
if equ(q, 0)  -- this means p is a root
    then t0 := part(2*(r**(1/2)))
    else t0 := aCubic(1, -p, -4*r, 4*p*r - q**2)
rt := part(t0 - p)**(1/2)
slist := append( quadratic( 1, rt, (-q/rt + t0)/2::F ),
                quadratic( 1, -rt, ( q/rt + t0)/2::F ))
-- Translate back:
    [s - a1/4::F for s in slist]

aQuartic(c4, c3, c2, c1, c0) ==
    needLcoef c4; needChar0()
    (c0 = 0) => 0$F
    a1 := c3/c4; a2 := c2/c4; a3 := c1/c4; a4 := c0/c4
    (a1 = 0 and a2 = 0 and a3 = 0) => (-a4)**(1/4)
p := part(a2-3*a1*a1/8::F)
q := part(a3-a1*a2/2::F + a1**2*a1/8::F)
r := part(a4-a1*a3/4::F + a1**2*a2/16::F - 3*a1**4/256::F)
if equ(q, 0)
    then t0 := part(2*(r**(1/2)))
    else t0 := aCubic(1, -p, -4*r, 4*p*r - q**2)
rt := part(t0 - p)**(1/2)
s := aQuadratic( 1, rt, (-q/rt + t0)/2::F )
s - a1/4::F

---------

— SOLVEFOR.dotabb —

"SOLVEFOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVEFOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SOLVEFOR" -> "PFECAT"

---------

package PSQFR PolynomialSquareFree

— PolynomialSquareFree.input —

)set break resume
)sys rm -f PolynomialSquareFree.output
)spool PolynomialSquareFree.output
)set message test on
)set message auto off
)clear all
---S 1 of 1
)show PolynomialSquareFree
--E 1

)spool
)lisp (bye)

---

— PolynomialSquareFree.help —

====================================================================
PolynomialSquareFree examples
====================================================================

This package computes square-free decomposition of multivariate polynomials over a coefficient ring which is an arbitrary gcd domain. The requirement on the coefficient domain guarantees that the content can be removed so that factors will be primitive as well as square-free. Over an infinite ring of finite characteristic, it may not be possible to guarantee that the factors are square-free.

See Also:
o )show PolynomialSquareFree

---

PolynomialSquareFree (PSQFR)

Exports:
squareFree
--- package PSQFR PolynomialSquareFree ---

)abbrev package PSQFR PolynomialSquareFree
++ Date Last Updated: November 1993, (P.Gianni)
++ Description:
++ This package computes square-free decomposition of multivariate
++ polynomials over a coefficient ring which is an arbitrary gcd domain.
++ The requirement on the coefficient domain guarantees that the
++ \spadfun{content} can be
++ removed so that factors will be primitive as well as square-free.
++ Over an infinite ring of finite characteristic, it may not be possible to
++ guarantee that the factors are square-free.

PolynomialSquareFree(VarSet:OrderedSet,E,RC:GcdDomain,P):C == T where
  E:OrderedAbelianMonoidSup
  P:PolynomialCategory(RC,E,VarSet)

C == with
  squareFree : P -> Factored P
    ++ squareFree(p) returns the square-free factorization of the
    ++ polynomial p. Each factor has no repeated roots, and the
    ++ factors are pairwise relatively prime.

T == add
  SUP ==> SparseUnivariatePolynomial(P)
  NNI  ==> NonNegativeInteger
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FF    ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

finSqFr : (P,List VarSet) -> Factored P
pthPower : P -> Factored P
pPolRoot : P -> P
putPth : P -> P

chrc:=characteristic$RC

if RC has CharacteristicNonZero then
  -- find the p-th root of a polynomial
  pPolRoot(f:P) : P ==
    lvar:=variables f
    empty? lvar => f
    mv:=first lvar
    uf:=univariate(f,mv)
    uf:=divideExponents(uf,chrc)::SUP
    uf:=map(pPolRoot,uf)
    multivariate(uf,mv)

  -- substitute variables with their p-th power
  putPth(f:P) : P ==
lvar := variables f
empty? lvar => f
mv := first lvar
uf := univariate(f, mv)
uf := multiplyExponents(uf, chrc)::SUP
uf := map(putPth, uf)
multivariate(uf, mv)

-- the polynomial is a perfect power
pthPower(f:P) : Factored P ==
proot := 0
isSq := false
if (g := charthRoot f) case "failed" then proot := pPolRoot(f)
else
proot := g :: P
isSq := true
psqfr := finSqFr(proot, variables f)
isSq =>
makeFR((unit psqfr)**chrc, [[u.flg, u.fctr, (u.xpnt)*chrc] for u in factorList psqfr])
makeFR((unit psqfr), ["nil", putPth u.fctr, u.xpnt] for u in factorList psqfr)

-- compute the square free decomposition, finite characteristic case
finSqFr(f:P, lvar: List VarSet) : Factored P ==
empty? lvar => pthPower(f)
mv := first lvar
lvar := lvar.rest
differentiate(f, mv) = 0 => finSqFr(f, lvar)
uf := univariate(f, mv)
cont := content uf
cont1 := 1
uf := (uf exquo cont)::SUP
squf := squareFree(uf)$UnivariatePolynomialSquareFree(P, SUP)
pfaclist := List FF := []
for u in factorList squf repeat
uexp := NNI := (u.xpnt)::NNI
u.flg = "sqfr" => -- the square free factor is OK
pfaclist := cons([u.flg, multivariate(u.fctr, mv), uexp], pfaclist)
else
listfin1 := finSqFr(multivariate(u.fctr, mv), lvar)
listfin1 := squareFree multivariate(u.fctr, mv)
flistfin1 := [[uu.flg, uu.fctr, uu.xpnt*uexp] for uu in factorList listfin1]
cont1 := cont1*([unit listfin1]**uexp)
pfaclist := append([listfin1, pfaclist])
kont := cont*cont1
cont := 1 =>
sq := squareFree cont
pfaclist := append(factorList sq, pfaclist)
```lisp
makeFR(unit(sqp)*coefficient(unit squf,0),pfaclist)
makeFR(coefficient(unit squf,0),pfaclist)

squareFree(p:P) ==
  mv:=mainVariable p
  mv case "failed" => makeFR(p,[])$Factored(P)
  characteristic$RC ^=0 => finSqFr(p,variables p)
  up:=univariate(p,mv)
  cont := content up
  up := (up exquo cont)::SUP
  squp := squareFree(up)$UnivariatePolynomialSquareFree(P,SUP)
  pfaclist:List FF :=
    [[u.flg,multivariate(u.fctr,mv),u.xpnt]
          for u in factorList squp]
  cont ^= 1 =>
    sqp := squareFree cont
    makeFR(unit(sqp)*coefficient(unit squp,0),
      append(factorList sqp, pfaclist))
  makeFR(coefficient(unit squp,0),pfaclist)
```

---

--- PSQFR.dotabb ---

"PSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PSQFR" -> "PFECAT"

---

package POLY2UP PolynomialToUnivariatePolynomial

--- PolynomialToUnivariatePolynomial.input ---

)set break resume
)sys rm -f PolynomialToUnivariatePolynomial.output
)spool PolynomialToUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialToUnivariatePolynomial
--E 1
This package is primarily to help the interpreter do coercions. It allows you to view a polynomial as a univariate polynomial in one of its variables with coefficients which are again a polynomial in all the other variables.

See Also:
- )show PolynomialToUnivariatePolynomial
++ It allows you to view a polynomial as a
++ univariate polynomial in one of its variables with
++ coefficients which are again a polynomial in all the
++ other variables.

PolynomialToUnivariate Polynomial(x: Symbol, R: Ring): with
   univariate: (Polynomial R, Variable x) ->
      Univariate Polynomial(x, Polynomial R)
   ++ univariate(p, x) converts the polynomial p to a one of type
   ++ \spad{Univariate Polynomial(x, Polynomial(R))}, i.e. as a member of \spad{R[...][x]}.
   == add
      univariate(p, y) ==
      q:Sparse Univariate Polynomial(Polynomial R) := univariate(p, x)
      map(x1+->x1, q)$Univariate PolynomialCategoryFunctions2(Polynomial R, Sparse Univariate Polynomial Polynomial R, Polynomial R, Univariate Polynomial(x, Polynomial R))
PowerSeriesLimitPackage examples
====================================================================

PowerSeriesLimitPackage implements limits of expressions in one or more variables as one of the variables approaches a limiting value. Included are two-sided limits, left- and right-hand limits, and limits at plus or minus infinity.

See Also:
o )show PowerSeriesLimitPackage

PowerSeriesLimitPackage (LIMITPS)

Exports:
complexLimit limit

--- package LIMITPS PowerSeriesLimitPackage ---

)abbrev package LIMITPS PowerSeriesLimitPackage
++ Author: Clifton J. Williamson
++ Date Created: 21 March 1989
++ Date Last Updated: 30 March 1994
++ Description:
++ PowerSeriesLimitPackage implements limits of expressions
++ in one or more variables as one of the variables approaches a
++ limiting value. Included are two-sided limits, left- and right-
++ hand limits, and limits at plus or minus infinity.

\textbf{PowerSeriesLimitPackage(R,FE):} \textbf{Exports == Implementation} where

\textbf{R :} \textbf{Join(GcdDomain,OrderedSet,RetractableTo Integer,}_
\textbf{LinearlyExplicitRingOver Integer)}

\textbf{FE :} \textbf{Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,}_
\textbf{FunctionSpace R)}

\textbf{Z} \textbf{==> Integer}

\textbf{RN} \textbf{==> Fraction Integer}

\textbf{RF} \textbf{==> Fraction Polynomial R}

\textbf{OFE} \textbf{==> OrderedCompletion FE}

\textbf{OPF} \textbf{==> OnePointCompletion FE}

\textbf{SY} \textbf{==> Symbol}

\textbf{EQ} \textbf{==> Equation}

\textbf{LF} \textbf{==> LiouvillianFunction}

\textbf{UTS} \textbf{==> UnivariateTaylorSeries}

\textbf{ULS} \textbf{==> UnivariateLaurentSeries}

\textbf{UPXS} \textbf{==> UnivariatePuiseuxSeries}

\textbf{EFULS} \textbf{==> ElementaryFunctionsUnivariateTaylorSeries}

\textbf{EFUPXS} \textbf{==> ElementaryFunctionsUnivariatePuiseuxSeries}

\textbf{FS2UPS} \textbf{==> FunctionSpaceToUnivariatePowerSeries}

\textbf{FS2EXPXP} \textbf{==> FunctionSpaceToExponentialExpansion}

\textbf{Problem} \textbf{==> Record(func:String,prob:String)}

\textbf{RESULT} \textbf{==> Union(OFE,"failed"})

\textbf{TwoSide} \textbf{==> Record(leftHandLimit:RESULT,rightHandLimit:RESULT)}

\textbf{U} \textbf{==> Union(OFE,TwoSide,"failed")}

\textbf{SIGNEF} \textbf{==> ElementaryFunctionSign(R,FE)}

\textbf{Exports == with}

\textbf{limit: (FE,EQ OFE) -> U}
\textbf{++ limit(f(x),x = a) computes the real limit \texttt{lim(x -> a,f(x))}.}

\textbf{complexLimit: (FE,EQ OPF) -> Union(OPF,"failed")}
\textbf{++ complexLimit(f(x),x = a) computes the complex limit}
\textbf{++ \texttt{lim(x -> a,f(x))}.}

\textbf{limit: (FE,EQ FE,String) -> RESULT}
\textbf{++ limit(f(x),x=a,"left") computes the left hand real limit}
\textbf{++ \texttt{lim(x -> a-,f(x))};}
\textbf{++ \texttt{lim(x -> a+,f(x))}} computes the right hand real limit
\textbf{++ \texttt{lim(x -> a+,f(x))}.}

\textbf{Implementation == add}
\textbf{import ToolsForSign(R)}
\textbf{import ElementaryFunctionStructurePackage(R,FE)}

\textbf{zeroFE:FE := 0}
\textbf{anyRootsOrA trium? : FE -> Boolean}
\textbf{complLimit : (FE,SY) -> Union(OPF,"failed")}
okProblem? : (String,String) -> Boolean
realLimit : (FE,SY) -> U
xxpLimit : (FE,SY) -> RESULT
limitPlus : (FE,SY) -> RESULT
localsubst : (FE,Kernel FE,Z,FE) -> FE
locallimit : (FE,SY,DFE) -> U
locallimitcomplex : (FE,SY,OPF) -> Union(OPF,"failed")
poleLimit:(RN,FE,SY) -> U
poleLimitPlus:(RN,FE,SY) -> RESULT

noX?: (FE,SY) -> Boolean
noX?(fcn,x) == not member?(x,variables fcn)

constant?: FE -> Boolean
constant? fcn == empty? variables fcn

firstNonLogPtr: (FE,SY) -> List Kernel FE
firstNonLogPtr(fcn,x) ==
  -- returns a pointer to the first element of kernels(fcn) which
  -- has 'x' as a variable, which is not a logarithm, and which is
  -- not simply 'x'
  list := kernels fcn
  while not empty? list repeat
    ker := first list
    not is?(ker,"log" :: Symbol) and member?(x,variables(ker::FE)) and
      not(x = name(ker)) =>
      return list
    list := rest list
  empty()

finiteValueAtInfinity?: Kernel FE -> Boolean
finiteValueAtInfinity? ker ==
  is?(ker,"erf" :: Symbol) => true
  is?(ker,"sech" :: Symbol) => true
  is?(ker,"csch" :: Symbol) => true
  is?(ker,"tanh" :: Symbol) => true
  is?(ker,"coth" :: Symbol) => true
  is?(ker,"atan" :: Symbol) => true
  is?(ker,"acot" :: Symbol) => true
  is?(ker,"asec" :: Symbol) => true
  is?(ker,"acsc" :: Symbol) => true
  is?(ker,"acsch" :: Symbol) => true
  is?(ker,"acoth" :: Symbol) => true
  is?(ker,"fresnelS" :: Symbol) => true
  is?(ker,"fresnelC" :: Symbol) => true
  error "finiteValueAtInfinity? true, but unknown value at infinity"

knownValueAtInfinity?: Kernel FE -> Boolean
knownValueAtInfinity? ker ==
  is?(ker,"exp" :: Symbol) => true
is?(ker,"sinh" :: Symbol) => true
is?(ker,"cosh" :: Symbol) => true
false

leftOrRight: (FE,SY,FE) -> SingleInteger
leftOrRight(fcn,x,limVal) ==
  -- function is called when limitPlus(fcn,x) = limVal
  -- determines whether the limiting value is approached
  -- from the left or from the right
  (value := limitPlus(inv(fcn - limVal),x)) case "failed" => 0
  (inf := whatInfinity(val := value :: OFE)) = 0 =>
    error "limit package: internal error"
  inf

specialLimit1: (FE,SY) -> RESULT
specialLimitKernel: (Kernel FE,SY) -> RESULT
specialLimitNormalize: (FE,SY) -> RESULT
specialLimit: (FE, SY) -> RESULT

specialLimit(fcn, x) ==
xkers := [k for k in kernels fcn | member?(x,variables(k::FE))]
#xkers = 1 => specialLimit1(fcn,x)
um := numerator fcn
den := denominator fcn
for k in xkers repeat
  (fval := limitPlus(k::FE,x)) case "failed" =>
    return specialLimitNormalize(fcn,x)
  whatInfinity(val := fval::OFE) ^= 0 =>
    return specialLimitNormalize(fcn,x)
  (valu := retractIfCan(val)@Union(FE,"failed")) case "failed" =>
    return specialLimitNormalize(fcn,x)
  finVal := valu :: FE
  num := eval(num, k, finVal)
  den := eval(den, k, finVal)
  den = 0 => return specialLimitNormalize(fcn,x)
  (num/den) :: OFE :: RESULT

specialLimitNormalize(fcn,x) == -- tries to normalize result first
  nfcn := normalize(fcn)
  fcn ^= nfcn => limitPlus(nfcn,x)
xkers := [k for k in tower fcn | member?(x,variables(k::FE))]
  # xkers ^= 2 => "failed"
  expKers := [k for k in xkers | is?(k, "exp" :: Symbol)]
  # expKers ^= 1 => "failed"
  -- fcn is a rational function of x and exp(g(x)) for some rational function g
  expKer := first expKers
  (fval := limitPlus(expKer::FE,x)) case "failed" => "failed"
  vv := new()$SY; eq : EQ FE := equation(expKer :: FE,vv :: FE)
  cc := eval(fcn,eq)
  expKerLim := fval :: OFE
-- following test for "failed" is needed due to compiler bug
-- limVal case OFE generates EQCAR(limVal, 1) which fails on atom "failed"
(limVal := locallimit(cc,vv,expKerLim)) case "failed" => "failed"
limVal case OFE =>
  limm := limVal :: OFE
  (lim := retractIfCan(limm)@Union(FE,"failed")) case "failed" =>
    "failed" -- need special handling for directions at infinity
limitPlus(lim, x)
"failed"

-- limit of expression having only 1 kernel involving x
specialLimit1(fcn,x) ==
  -- find the first interesting kernel in tower(fcn)
xkers := [k for k in kernels fcn | member?(x,variables(k::FE))]
#xkers ^= 1 => "failed"
ker := first xkers
vv := new()$SY; eq : EQ FE := equation(ker :: FE,vv :: FE)
cc := eval(fcn,eq)
member?(x,variables cc) => "failed"
(lim := specialLimitKernel(ker, x)) case "failed" => lim
argLim : OFE := lim :: OFE
(limVal := locallimit(cc,vv,argLim)) case "failed" => "failed"
limVal case OFE => limVal :: OFE
"failed"

-- limit of single kernel involving x
specialLimitKernel(ker,x) ==
  is?(ker,"log" :: Symbol) =>
    args := argument ker
    empty? args => "failed" -- error "No argument"
    not empty? rest args => "failed" -- error "Too many arguments"
    arg := first args
    -- compute limit(x -> 0+,arg)
    (limm := limitPlus(arg,x)) case "failed" => "failed"
    lim := limm :: OFE
    (inf := whatInfinity lim) = -1 => "failed"
    argLim : OFE :=
      -- log(+infinity) = +infinity
      inf = 1 => lim
      -- now 'lim' must be finite
      (li := retractIfCan(lim)@Union(FE,"failed")) :: FE = 0 =>
        -- log(0) = -infinity
        leftOrRight(arg,x,0) = 1 => minusInfinity()
        return "failed"
      log(li) :: OFE
    -- kernel should be a function of one argument f(arg)
    args := argument(ker)
    empty? args => "failed" -- error "No argument"
    not empty? rest args => "failed" -- error "Too many arguments"
    arg := first args
-- compute \( \lim_{x \to 0^+} \text{arg} \)
(limm := limitPlus(arg,x)) case "failed" => "failed"
lim := limm :: OFE
f := elt(operator ker,(var := new($)SY) :: FE)
-- compute \( \lim_{x \to 0^+} f(\text{arg}) \)
-- case where 'lim' is finite
(inf := whatInfinity lim) = 0 =>
  is?(ker,"erf" :: Symbol) => erf(retract(lim)@FE)$LF(R,FE) :: OFE
  kerValue case OFE => kerValue :: OFE
  "failed"
-- case where 'lim' is plus infinity
inf = 1 =>
  finiteValueAtInfinity? ker =>
    val : FE :=
      is?(ker,"erf" :: Symbol) => 1
      is?(ker,"sech" :: Symbol) => 0
      is?(ker,"csch" :: Symbol) => 0
      is?(ker,"tanh" :: Symbol) => 0
      is?(ker,"coth" :: Symbol) => 0
      is?(ker,"atan" :: Symbol) => pi()/(2 :: FE)
      is?(ker,"acot" :: Symbol) => 0
      is?(ker,"asec" :: Symbol) => pi()/(2 :: FE)
      is?(ker,"acsc" :: Symbol) => 0
      is?(ker,"acsch" :: Symbol) => 0
      is?(ker,"fresnelS" :: Symbol) => -sqrt(pi()/(8::FE))
      is?(ker,"fresnelC" :: Symbol) => -sqrt(pi()/(8::FE))
      error "finiteValueAtInfinity? true, but unknown value at infinity"
    -- ker must be acoth
    0
    val :: OFE
  knownValueAtInfinity? ker =>
    lim -- \( \lim_{x \to \infty} \text{exp, cosh, sinh} \) = inf
    "failed"
-- case where 'lim' is minus infinity
finiteValueAtInfinity? ker =>
  val : FE :=
    is?(ker,"erf" :: Symbol) => -1
    is?(ker,"sech" :: Symbol) => 0
    is?(ker,"csch" :: Symbol) => 0
    is?(ker,"tanh" :: Symbol) => 0
    is?(ker,"coth" :: Symbol) => 0
    is?(ker,"atan" :: Symbol) => -pi()/(2 :: FE)
    is?(ker,"acot" :: Symbol) => pi()
    is?(ker,"asec" :: Symbol) => -pi()/(2 :: FE)
    is?(ker,"acsc" :: Symbol) => -pi()
    is?(ker,"acsch" :: Symbol) => 0
    -- ker must be acoth
    0
  val :: OFE
knownValueAtInfinity? ker =>
   is?(ker,"exp" :: Symbol) => (0@FE) :: OFE
   is?(ker,"sinh" :: Symbol) => lim
   is?(ker,"cosh" :: Symbol) => plusInfinity()
   "failed"
   "failed"

logOnlyLimit: (FE,SY) -> RESULT
logOnlyLimit(coef,x) ==
   -- this function is called when the 'constant' coefficient involves
   -- the variable 'x'. Its purpose is to compute a right hand limit
   -- of an expression involving log x. Here log x is replaced by -1/v,
   -- where v is a new variable. If the new expression no longer involves
   -- x, then take the right hand limit as v -> 0+
   vv := new()$SY
   eq : EQ FE := equation(log(x :: FE),-inv(vv :: FE))
   member?(x,variables(cc := eval(coef,eq))) => "failed"
   limitPlus(cc,vv)

locallimit(fcn,x,a) ==
   -- Here 'fcn' is a function f(x) = f(x,...) in 'x' and possibly
   -- other variables, and 'a' is a limiting value. The function
   -- computes lim(x -> a,f(x)).
   xK := retract(x::FE)@Kernel(FE)
   (n := whatInfinity a) = 0 =>
      realLimit(localsubst(fcn,xK,1,retract(a)@FE),x)
   (u := limitPlus(eval(fcn,xK,n * inv(xK::FE)),x))
   case "failed" => "failed"
   u::OFE

localsubst(fcn, k, n, a) ==
   a = 0 and n = 1 => fcn
   eval(fcn,k,n * (k::FE) + a)

locallimitcomplex(fcn,x,a) ==
   xK := retract(x::FE)@Kernel(FE)
   (g := retractIfCan(a)@Union(FE,"failed")) case FE =>
      complLimit(localsubst(fcn,xK,1,g::FE),x)
   complLimit(eval(fcn,xK,inv(xK::FE)),x)

limit(fcn,eq,str) ==
   (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
   x := xx :: SY; a := rhs eq
   xK := retract(x::FE)@Kernel(FE)
   limitPlus(localsubst(fcn,xK,direction str,a),x)

anyRootsDrAtigs? fcn ==
   -- determines if 'fcn' has any kernels which are roots
   -- or if 'fcn' has any kernels which are inverse trig functions
-- which could produce series expansions with fractional exponents
for kernel in tower fcn repeat
  is?(kernel,"nthRoot" :: Symbol) => return true
  is?(kernel,"asin" :: Symbol) => return true
  is?(kernel,"acos" :: Symbol) => return true
  is?(kernel,"asec" :: Symbol) => return true
  is?(kernel,"acsc" :: Symbol) => return true
false

complLimit(fcn,x) ==
-- computes lim(x -> 0,fcn) using a Puiseux expansion of fcn,
-- if fcn is an expression involving roots, and using a Laurent
-- expansion of fcn otherwise
lim : FE :=
  anyRootsOrAtrigs? fcn =>
    ppack := FS2UPS(R,FE,RN,_.
      UPXS(FE.x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_.
      EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
    pseries := exprToUPS(fcn,false,"complex")$ppack
    pseries case %problem => return "failed"
    if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
    pole? upxs => return infinity()
    coefficient(upxs,0)
  lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_.
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
  lseries := exprToUPS(fcn,false,"complex")$lpack
  lseries case %problem => return "failed"
  if pole?(uls := lseries.%series) then uls := map(normalize,uls)
  pole? uls => return infinity()
  coefficient(uls,0)
  -- can the following happen?
  member?(x,variables lim) =>
    member?(x,variables(answer := normalize lim)) =>
      error "limit: can't evaluate limit"
  answer :: OPF
  lim :: FE :: OPF

okProblem?(function,problem) ==
  (function = "log") or (function = "nth root") =>
    (problem = "series of non-zero order") or _
    (problem = "negative leading coefficient")
  (function = "atan") => problem = "branch problem"
  (function = "erf") => problem = "unknown kernel"
  problem = "essential singularity"

poleLimit(order,coef,x) ==
-- compute limit for function with pole
not member?(x,variables coef) =>
  (s := sign(coef)$SIGNEF) case Integer =>
    rtLim := (s :: Integer) * plusInfinity()
even? numer order => rtLim
even? denom order => ["failed",rtLim]\$TwoSide
[\!-rtLim,rtLim]\$TwoSide
-- infinite limit, but cannot determine sign
"failed"
error "limit: can't evaluate limit"

poleLimitPlus(order,coef,x) ==
-- compute right hand limit for function with pole
not member?(x,variables coef) =>
  (s := sign(coef)$SIGNEF) case Integer =>
  (s :: Integer) * plusInfinity()
-- infinite limit, but cannot determine sign
"failed"
(clim := specialLimit(coef,x)) case "failed" => "failed"
zero? (lim := clim :: OFE) =>
-- in this event, we need to determine if the limit of
-- the coef is 0+ or 0-
  (cclim := specialLimit(inv coef,x)) case "failed" => "failed"
  ss := whatInfinity(cclim :: OFE) :: Z
  zero? ss =>
    error "limit: internal error"
  ss * plusInfinity()
  t := whatInfinity(lim :: OFE) :: Z
  zero? t =>
    (tt := sign(coef)$SIGNEF) case Integer =>
    (tt :: Integer) * plusInfinity()
-- infinite limit, but cannot determine sign
"failed"
  t * plusInfinity()

realLimit(fcn,x) ==
-- computes lim(x -> 0,fcn) using a Puiseux expansion of fcn,
-- if fcn is an expression involving roots, and using a Laurent
-- expansion of fcn otherwise
lim : Union(FE,"failed") :=
anyRootsOrAtrigs? fcn =>
  ppack := FS2UPS(R,FE,NN,_,
    UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_,
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
  pseries := exprToUPS(fcn,true,"real: two sides")$ppack
  pseries case %problem =>
    trouble := pseries.%problem
    function := trouble.func; problem := trouble.prob
    okProblem?(function,problem) =>
      left :=
        xK : Kernel FE := kernel x
        fcn0 := eval(fcn,xK,-(xK :: FE))
        limitPlus(fcn0,x)
      right := limitPlus(fcn,x)
(left case "failed") and (right case "failed") =>
  return "failed"
if (left case OFE) and (right case OFE) then
  (left :: OFE) = (right :: OFE) => return (left :: OFE)
return ([left,right]$TwoSide)
return "failed"
if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
pole? upxs =>
  cp := coefficient(upxs,ordp := order upxs)
  return poleLimit(ordp,cp,x)
coefficient(upxs,0)
lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
      EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
lseries := exprToUPS(fcn,true,"real: two sides")$lpack
lseries case %problem =>
  trouble := lseries.%problem
  function := trouble.func; problem := trouble.prob
  okProblem?(function,problem) =>
    left :=
      xK : Kernel FE := kernel x
      fcn0 := eval(fcn,xK,-(xK :: FE))
      limitPlus(fcn0,x)
    right := limitPlus(fcn,x)
  (left case "failed") and (right case "failed") =>
    return "failed"
if (left case OFE) and (right case OFE) then
  (left :: OFE) = (right :: OFE) => return (left :: OFE)
return ([left,right]$TwoSide)
return "failed"
if pole?(uls := lseries.%series) then uls := map(normalize,uls)
pole? uls =>
  cl := coefficient(uls,ordl := order uls)
  return poleLimit(ordl :: RN,cl,x)
coefficient(uls,0)
lim case "failed" => "failed"
member?(x,variables(lim :: FE)) =>
  member?(x,variables(answer := normalize(lim :: FE))) =>
    error "limit: can't evaluate limit"
answer :: OFE
lim :: FE :: OFE

xxpLimit(fcn,x) ==
  -- computes lim(x -> 0+,fcn) using an exponential expansion of fcn
  xpack := FS2EXPXP(R,FE,x,zeroFE)
  xxp := exprToXXP(fcn,true)$xpack
  xxp case %problem => "failed"
  limitPlus(xxp.%expansion)

limitPlus(fcn,x) ==
  -- computes lim(x -> 0+,fcn) using a generalized Puiseux expansion
-- of fcn, if fcn is an expression involving roots, and using a
-- generalized Laurent expansion of fcn otherwise
lim : Union(FE,"failed") :=
  anyRootsOrAtrigs? fcn =>
    ppack := FS2UPS(R,FE,RN,_
      UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_
      EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
    pseries := exprToGenUPS(fcn,true,"real: right side")$ppack
    pseries case %problem =>
      trouble := pseries.%problem
      ff := trouble.func; pp := trouble.prob
      (pp = "negative leading coefficient") => return "failed"
      "failed"
    -- pseries case %problem => return "failed"
    if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
    pole7 upxs =>
      cp := coefficient(upxs,ordp := order upxs)
      return poleLimitPlus(ordp,cp,x)
    coefficient(upxs,0)
  lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
  lseries := exprToGenUPS(fcn,true,"real: right side")$lpack
  lseries case %problem =>
    trouble := lseries.%problem
    ff := trouble.func; pp := trouble.prob
    (pp = "negative leading coefficient") => return "failed"
    "failed"
  -- lseries case %problem => return "failed"
  if pole?(uls := lseries.%series) then uls := map(normalize,uls)
  pole7 uls =>
    cl := coefficient(uls,ordl := order uls)
    return poleLimitPlus(ordl :: RN,cl,x)
  coefficient(uls,0)
lim case "failed" =>
  (xLim := xxpLimit(fcn,x)) case "failed" => specialLimit(fcn,x)
xLim
member?(x:variables(lim :: FE)) =>
  member?(x:variables(answer := normalize(lim :: FE))) =>
  (xLim := xxpLimit(answer,x)) case "failed" => specialLimit(answer,x)
xLim
answer :: OFE
lim :: FE :: OFE

limit(fcn:FE,eq:EQ OFE) ==
  (f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
    error "limit:left hand side must be a variable"
  (xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
    error "limit:left hand side must be a variable"
  x := xx :: SY; a := rhs eq
locallimit(fcn,x,a)
complexLimit(fcn:FE, eq: EQ OPF) ==
(f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
   error "limit:left hand side must be a variable"
(xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
   error "limit:left hand side must be a variable"
x := xx :: SY; a := rhs eq
locallimitcomplex(fcn,x,a)

— LIMITPS.dotabb —

"LIMITPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIMITPS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"LIMITPS" -> "ULSCCAT"

package PREASSOC PrecomputedAssociatedEquations

— PrecomputedAssociatedEquations.input —

)set break resume
)sys rm -f PrecomputedAssociatedEquations.output
)spool PrecomputedAssociatedEquations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrecomputedAssociatedEquations
--E 1

)spool
)lisp (bye)

— PrecomputedAssociatedEquations.help —

================================================================================
PrecomputedAssociatedEquations examples
PrecomputedAssociatedEquations stores some generic precomputations which speed up the computations of the associated equations needed for factoring operators.

See Also:
o )show PrecomputedAssociatedEquations

---

PrecomputedAssociatedEquations (PREASSOC)

Exports:
firstUncouplingMatrix

— package PREASSOC PrecomputedAssociatedEquations —

)abbrev package PREASSOC PrecomputedAssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 13 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{PrecomputedAssociatedEquations} stores some generic
++ precomputations which speed up the computations of the
++ associated equations needed for factoring operators.

PrecomputedAssociatedEquations(R, L):Exports == Implementation where
R: IntegralDomain
L: LinearOrdinaryDifferentialOperatorCategory R
PI ==> PositiveInteger
N ==> NonNegativeInteger
\( A \Rightarrow \text{PrimitiveArray R} \)

\( U \Rightarrow \text{Union(\text{Matrix R}, \text{"failed"})} \)

Exports => with

\( \text{firstUncouplingMatrix: (L, PI)} \Rightarrow U \)

++ \text{firstUncouplingMatrix(op, m)} returns the matrix A such that
++ \text{\texttt{\textbackslash spad\{A w = (W',W'',...,W^N)\}} in the corresponding associated}
++ \text{equations for right-factors of order m of op.}
++ \text{Returns "failed" if the matrix A has not been precomputed for}
++ \text{the particular combination \texttt{\textbackslash spad\{degree(L), m\}}.}

Implementation => add

A32: L \rightarrow U

A42: L \rightarrow U

A425: (A, A, A) \rightarrow \text{List R}

A426: (A, A, A) \rightarrow \text{List R}

\( \text{makeMonic: L \rightarrow Union(A, \text{"failed"})} \)

diff:L := D()

\( \text{firstUncouplingMatrix(op, m)} =\)
\n\( \text{n := degree op} \)
\n\( \text{n = 3 and m = 2 \Rightarrow A32 op} \)
\n\( \text{n = 4 and m = 2 \Rightarrow A42 op} \)
\n"failed"

\( \text{makeMonic op =} \)
\n\( \text{lc := leadingCoefficient op} \)
\n\( \text{a:A := new(n := degree op, 0)} \)
\n\( \text{for i in 0..(n-1)::N repeat} \)
\n\( (u := \text{coefficient(op, i) exquo lc}) \text{ case "failed" \Rightarrow return "failed"} \)
\n\( \text{a.i := - (u::R)} \)
\n\( \text{a} \)

A32 op =
\( \text{(u := makeMonic op) case "failed" \Rightarrow "failed"} \)
\( \text{a := u::A} \)
\( \text{matrix [[0, 1, 0], [a.1, a.2, 1],} \)
\( \text{[diff(a.1) + a.1 * a.2 - a.0, diff(a.2) + a.2**2 + a.1, 2 * a.2]}} \)

A42 op =
\( \text{(u := makeMonic op) case "failed" \Rightarrow "failed"} \)
\( \text{a := u::A} \)
\( \text{a':A := new(4, 0)} \)
\( \text{a'':A := new(4, 0)} \)
\( \text{for i in 0..3 repeat} \)
\( \text{a'.i := diff(a.i)} \)
\( \text{a''.i := diff(a'.i)} \)
\( \text{matrix [[0, 1, 0, 0, 0], [0, 0, 1, 1, 0, 0], [a.1,a.2,0,a.3,2::R,0],} \)
\( \text{[a'.1 + a.1 * a.3 - 2 * a.0, a'.2 + a.2 * a.3 + a.1, 3 * a.2,} \)
A425\(a, a', a''\) ==
\[
\begin{align*}
\text{diff}(a'').1 &+ 3 \cdot a.1 \cdot a'''.3 + a.3 \cdot a'''.1 - 2 \cdot a'''.0 \\
+ (3 \cdot a'.1 + 5 \cdot a.1 \cdot a.3 - 7 \cdot a.0) \cdot a'''.3 + 3 \cdot a.1 \cdot a''.2 \\
+ (a.3 \cdot a'''.2 + a.3 \cdot a'''.3 + 3 \cdot a.1) \cdot a'''.3 + 3 \cdot a.1 \cdot a'''.2 \\
+ 2 \cdot a.2 \cdot a.''.1 - 6 \cdot a'.0 + a.2 \cdot a.3 \cdot a.1 \cdot a.3 \cdot a.2 \\
+ (2 \cdot a.2 \cdot a.3 + 2 \cdot a.1 \cdot a.2, \\
5 \cdot a'''.2 + 10 \cdot a.2 \cdot a.''.3 + 5 \cdot a.3 \cdot a'.2 + a'.1 \\
+ 5 \cdot a.2 \cdot a.3 \cdot a.2 + 4 \cdot a.1 \cdot a.3 + 5 \cdot a.2 \cdot a.2 + 4 \cdot a.0, \\
\text{diff}(a'''.3) + 4 \cdot a.3 \cdot a'.''.3 + 3 \cdot a''.2 + 3 \cdot a'.3 \cdot a'''.2 \\
+ (6 \cdot a.3 \cdot a.2 + 4 \cdot a.2) \cdot a.''.3 + 5 \cdot a.3 \cdot a'.2 + 3 \cdot a'.1 \\
+ a.3 \cdot a'.3 + 3 \cdot a.2 \cdot a.3 \cdot a'.2 + 2 \cdot a.1 \cdot a.3 + a.2 \cdot a.2 + 4 \cdot a.0, \\
5 \cdot a'..3 + 15 \cdot a.3 \cdot a'.3 + 10 \cdot a'.2 + 5 \cdot a.3 \cdot a.3 + 9 \cdot a.3 \cdot a.2 + 4 \cdot a.2
\end{align*}
\]

A426\(a, a', a''\) ==
\[
\begin{align*}
\text{diff}(a'').1 &+ 3 \cdot a.1 \cdot a'''.3 + a.3 \cdot a'''.1 - 2 \cdot a'''.0 \\
+ (3 \cdot a'.1 + 5 \cdot a.1 \cdot a.3 - 7 \cdot a.0) \cdot a'''.3 + 3 \cdot a.1 \cdot a''.2 \\
+ (a.3 \cdot a'''.2 + a.3 \cdot a'''.3 + 3 \cdot a.1) \cdot a'''.3 + 3 \cdot a.1 \cdot a'''.2 \\
+ 2 \cdot a.2 \cdot a.''.1 - 6 \cdot a'.0 + a.2 \cdot a.3 \cdot a.1 \cdot a.3 \cdot a.2 \\
+ (2 \cdot a.2 \cdot a.3 + 2 \cdot a.1 \cdot a.2, \\
5 \cdot a'''.2 + 10 \cdot a.2 \cdot a.''.3 + 5 \cdot a.3 \cdot a'.2 + a'.1 \\
+ 5 \cdot a.2 \cdot a.3 \cdot a.2 + 4 \cdot a.1 \cdot a.3 + 5 \cdot a.2 \cdot a.2 + 4 \cdot a.0, \\
\text{diff}(a'''.3) + 4 \cdot a.3 \cdot a'.''.3 + 3 \cdot a''.2 + 3 \cdot a'.3 \cdot a'''.2 \\
+ (6 \cdot a.3 \cdot a.2 + 4 \cdot a.2) \cdot a.''.3 + 5 \cdot a.3 \cdot a'.2 + 3 \cdot a'.1 \\
+ a.3 \cdot a'.3 + 3 \cdot a.2 \cdot a.3 \cdot a'.2 + 2 \cdot a.1 \cdot a.3 + a.2 \cdot a.2 + 4 \cdot a.0, \\
5 \cdot a'..3 + 15 \cdot a.3 \cdot a'.3 + 10 \cdot a'.2 + 5 \cdot a.3 \cdot a.3 + 9 \cdot a.3 \cdot a.2 + 4 \cdot a.2
\end{align*}
\]
This package provides tools for operating on primitive arrays with unary and binary functions involving different underlying types.

See Also:

- `)show PrimitiveArrayFunctions2`

---

PrimitiveArrayFunctions2 (PRIMARR2)

Exports:

map reduce scan
package PRIMARR2 PrimitiveArrayFunctions2

)abbrev package PRIMARR2 PrimitiveArrayFunctions2
++ Description:
++ This package provides tools for operating on primitive arrays
++ with unary and binary functions involving different underlying types

PrimitiveArrayFunctions2(A, B): Exports == Implementation where
A, B: Type

VA ==> PrimitiveArray A
VB ==> PrimitiveArray B
O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)
Exports ==> with
  scan : ((A, B) -> B, VA, B) -> VB
  ++ scan(f,a,r) successively applies
  ++ \texttt{reduce(f,x,r)} to more and more leading sub-arrays
  ++ of primitive array \texttt{a}.
  ++ More precisely, if \texttt{a} is \texttt{[a1,a2,...]}, then
  ++ \texttt{scan(f,a,r)} returns
  ++ \texttt{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
  ++
  ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
  ++X adder(a:Integer,b:Integer):Integer == a+b
  ++X scan(adder,[i for i in 1..10],0)$T1

  reduce : ((A, B) -> B, VA, B) -> B
  ++ reduce(f,a,r) applies function \texttt{f} to each
  ++ successive element of the
  ++ primitive array \texttt{a} and an accumulant initialized to \texttt{r}.
  ++ For example, \texttt{reduce(+$Integer,[1,2,3],0)}
  ++ does \texttt{3+(2+(1+0))}. Note that third argument \texttt{r}
  ++ may be regarded as the identity element for the function \texttt{f}.
  ++
  ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
  ++X adder(a:Integer,b:Integer):Integer == a+b
  ++X reduce(adder,[i for i in 1..10],0)$T1

  map : (A -> B, VA) -> VB
  ++ map(f,a) applies function \texttt{f} to each member of primitive array
  ++ \texttt{a} resulting in a new primitive array over \texttt{a}
  ++ possibly different underlying domain.
  ++
  ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
  ++X map(x+->x+2,[i for i in 1..10])$T1

Implementation ==> add
  map(f, v) == map(f, v)$O2
  scan(f, v, b) == scan(f, v, b)$O2
reduce(f, v, b) == reduce(f, v, b)$.O2

---

— PRIMARR2.dotabb —

"PRIMARR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMARR2"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"PRIMARR2" -> "A1AGG"

---

package PRIMELT PrimitiveElement

— PrimitiveElement.input —

)set break resume
)sys rm -f PrimitiveElement.output
)spool PrimitiveElement.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimitiveElement
--E 1

)spool
)lisp (bye)

---

— PrimitiveElement.help —

====================================================================
PrimitiveElement examples
====================================================================

PrimitiveElement provides functions to compute primitive elements
in algebraic extensions;

See Also:
 o )show PrimitiveElement
PrimitiveElement (PRIMELT)

Exports:
primitiveElement

— package PRIMELT PrimitiveElement —

)abbrev package PRIMELT PrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Description:
++ PrimitiveElement provides functions to compute primitive elements
++ in algebraic extensions;

PrimitiveElement(F): Exports == Implementation where
  F : Join(Field, CharacteristicZero)
  SY ==> Symbol
  P ==> Polynomial F
  UP ==> SparseUnivariatePolynomial F
  RC ==> Record(coef1: Integer, coef2: Integer, prim:UP)
  REC ==> Record(coef: List Integer, poly:List UP, prim: UP)

Exports ==> with
  primitiveElement: (P, SY, P, SY) -> RC
    ++ primitiveElement(p1, a1, p2, a2) returns \spad{[c1, c2, q]}
    ++ such that \spad{k(a1, a2) = k(a)}
    ++ where \spad{a = c1 a1 + c2 a2, and q(a) = 0}.
    ++ The pi's are the defining polynomials for the ai's.
    ++ The p2 may involve a1, but p1 must not involve a2.
    ++ This operation uses \spadfun{resultant}.  


primitiveElement: (List P, List SY) -> REC
  ++ primitiveElement([[p1,...,pn], [a1,...,an]]) returns
  ++ \spad{[[c1,...,cn], [q1,...,qn], q]}
  ++ such that then \spad{k(a1,...,an) = k(a)},
  ++ where \spad{a = a1 c1 + ... + an cn},
  ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
  ++ The pi’s are the defining polynomials for the ai’s.
  ++ This operation uses the technique of
  ++ \spadglossSee{groebner bases}{Groebner basis}.

primitiveElement: (List P, List SY, SY) -> REC
  ++ primitiveElement([[p1,...,pn], [a1,...,an], a]) returns
  ++ \spad{[[c1,...,cn], [q1,...,qn], q]}
  ++ such that then \spad{k(a1,...,an) = k(a)},
  ++ where \spad{a = a1 c1 + ... + an cn},
  ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
  ++ The pi’s are the defining polynomials for the ai’s.
  ++ This operation uses the technique of
  ++ \spadglossSee{groebner bases}{Groebner basis}.

Implementation ==> add
  import PolyGroebner(F)

multi : (UP, SY) -> P
randomInts: (NonNegativeInteger, NonNegativeInteger) -> List Integer
findUniv : (List P, SY, SY) -> Union(P, "failed")
incl? : (List SY, List SY) -> Boolean
triangularLinearIfCan:(List P,List SY,SY) -> Union(List UP,"failed")
innerPrimitiveElement: (List P, List SY, SY) -> REC

multi(p, v) == multivariate(map((f1:F):F +-> f1, p), v)
randomInts(n, m) == [symmetricRemainder(random()$Integer, m) for i in 1..n]
incl?(a, b) == every?((s1:SY):Boolean +-> member?(s1, b), a)
primitiveElement(l, v) == primitiveElement(l, v, new()$SY)

primitiveElement(p1, a1, p2, a2) ==
  -- one? degree(p2, a1) => [0, 1, univariate resultant(p1, p2, a1)]
  (degree(p2, a1) = 1) => [0, 1, univariate resultant(p1, p2, a1)]
  u := (new()$SY)::P
  b := a2::P
  for i in 10.. repeat
    c := symmetricRemainder(random()$Integer, i)
    w := u - c * b
    r := univariate resultant(eval(p1, a1, w), eval(p2, a1, w), a2)
    not zero? r and r = squareFreePart r => return [1, c, r]

findUniv(l, v, opt) ==
  for p in l repeat
    degree(p, v) > 0 and incl?(variables p, [v, opt]) => return p
  "failed"
triangleLinearIfCan(l, lv, w) ==
  (u := findUniv(l, w, w)) case "failed" => "failed"
  pw := univariate(u::P)
  ll := nil()$List(UP)
  for v in lv repeat
    ((u := findUniv(l, v, w)) case "failed") or
    (degree(p := univariate(u::P, v)) ^= 1) => return "failed"
    (bc := extendedEuclidean(univariate leadingCoefficient p, pw, 1))
      case "failed" => error "Should not happen"
    ll := concat(map((z1:F):F +-> z1,
                         (- univariate(coefficient(p,0)) * bc.coef1) rem pw), ll)
    concat(map((f1:F):F +-> f1, pw), reverse! ll)

primitiveElement(l, vars, uu) ==
  u := uu::P
  vv := [v::P for v in vars]
  elim := concat(vars, uu)
  w := uu::P
  n := #l
  for i in 10.. repeat
    cf := randomInts(n, i)
    (tt := triangularLinearIfCan(lexGroebner(
      concat(w - +/[c * t for c in cf for t in vv], l), elim),
        vars, uu)) case List(UP) =>
    ltt := tt::List(UP)
    return([cf, rest ltt, first ltt])
PrimitiveRatDE provides functions for in-field solutions of linear ordinary differential equations, in the transcendental case. The derivation to use is given by the parameter \( L \).

See Also:
- `)show PrimitiveRatDE`

---

**Exports:**
- `indicialEquation`
- `denomLODE`
- `indicialEquations`
- `splitDenominator`
++ Package ODEPRIM PrimitiveRatDE
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{PrimitiveRatDE} provides functions for in-field solutions of linear
++ ordinary differential equations, in the transcendental case.
++ The derivation to use is given by the parameter \spad{L}.

PrimitiveRatDE(F, UP, L, LQ): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  L : LinearOrdinaryDifferentialOperatorCategory UP
  LQ : LinearOrdinaryDifferentialOperatorCategory Fraction UP

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
UP2 ==> SparseUnivariatePolynomial UP
REC ==> Record(center:UP, equation:UP)

Exports ==> with
denomLODE: (L, RF) -> Union(UP, "failed")
  ++ denomLODE(op, g) returns a polynomial d such that
  ++ any rational solution of \spad{op y = g}
  ++ is of the form \spad{p/d} for some polynomial p, and
  ++ "failed", if the equation has no rational solution.
denomLODE: (L, List RF) -> UP
  ++ denomLODE(op, [g1,...,gm]) returns a polynomial
  ++ d such that any rational solution of \spad{op y = c1 g1 + ... + cm gm}
  ++ is of the form \spad{p/d} for some polynomial p.
indicialEquations: L -> List REC
  ++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
  ++ the \spad{d_i}'s are the affine singularities of \spad{op},
  ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
indicialEquations: (L, UP) -> List REC
  ++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
  ++ the \spad{d_i}'s are the affine singularities of \spad{op}
  ++ above the roots of \spad{p},
  ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
indicialEquation: (L, F) -> UP
  ++ indicialEquation(op, a) returns the indicial equation of \spad{op}
  ++ at \spad{a}.
indicialEquations: LQ -> List REC
  ++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
  ++ the \spad{d_i}'s are the affine singularities of \spad{op},
  ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}. 
indicialEquations: (LQ, UP) -> List REC
  ++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
  ++ the \spad{d_i}'s are the affine singularities of \spad{op}
  ++ above the roots of \spad{p},
  ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
indicialEquation: (LQ, F) -> UP
  ++ indicialEquation(op, a) returns the indicial equation of \spad{op}
  ++ at \spad{a}.
splitDenominator: (LQ, List RF) -> Record(eq:L, rh:List RF)
  ++ splitDenominator(op, [g1,...,gm]) returns \spad{op0, [h1,...,hm]}
  ++ such that the equations \spad{op y = c1 g1 + ... + cm gm} and
  ++ \spad{op0 y = c1 h1 + ... + cm hm} have the same solutions.

Implementation ==> add
import BoundIntegerRoots(F, UP)
import BalancedFactorisation(F, UP)
import InnerCommonDenominator(UP, RF, List UP, List RF)
import UnivariatePolynomialCategoryFunctions2(F, UP, UP, UP2)
tau : (UP, UP, UP, N) -> UP
NPbound : (UP, L, UP) -> N
denom : (L, UP, UP) -> UP
denom0 : (Z, L, UP, UP, UP) -> UP
indicialEq : (UP, List N, List UP) -> UP
separateZeros: (UP, UP) -> UP
UPfact : N -> UP
UP2UP2 : UP -> UP2
indeq : (UP, L) -> UP
NPmulambda : (UP, L) -> Record(mu:Z, lambda:List N, func:List UP)

diff := D()$L

UP2UP2 p == map((f1:F):UP +->f1::UP, p)
indicialEquations(op:L) == indicialEquations(op, leadingCoefficient op)
indicialEquation(op:L, a:F) == indeq(monomial(1, 1) - a::UP, op)
splitDenominator(op, lg) ==
  cd := splitDenominator coefficients op
  f := cd.den / gcd(cd.num)
  l:L := 0
  while op ^= 0 repeat
    l := l + monomial(retract(f * leadingCoefficient op), degree op)
    op := reductum op
  [l, [f * g for g in lg]]

tau(p, pp, q, n) ==
  ((pp ** n) * ((q exquo (p ** order(q, p)))::UP)) rem p

indicialEquations(op:LQ) ==
  indicialEquations(splitDenominator(op, empty()).eq)
indicialEquations(op:LQ, p:UP) ==
  indicialEquations(splitDenominator(op, empty()).eq, p)

indicialEquation(op:LQ, a:F) ==
  indeq(monomial(1, 1) - a::UP, splitDenominator(op, empty()).eq)

-- returns z(z-1)...(z-(n-1))
UPfact n ==
  zero? n => 1
  z := monomial(1, 1)$UP
  */[z - i::F::UP for i in 0..(n-1)::N]

indicialEq(c, lamb, lf) ==
  cp := diff c
  cc := UP2UP2 c
  s:UP2 := 0
  for i in lamb for f in lf repeat
    s := s + (UPfact i) * UP2UP2 tau(c, cp, f, i)
  primitivePart resultant(cc, s)

NPmulambda(c, l) ==
  lamb:List(N) := [d := degree l]
  lf:List(UP) := [a := leadingCoefficient l]
  mup := d::Z - order(a, c)
  while (l := reductum l) ^= 0 repeat
    a := leadingCoefficient l
    if (m := (d := degree l)::Z - order(a, c)) > mup then
      mup := m
      lamb := [d]
      lf := [a]
    else if (m = mup) then
      lamb := concat(d, lamb)
      lf := concat(a, lf)
    [mup, lamb, lf]

-- e = 0 means homogeneous equation
NPbound(c, l, e) ==
  rec := NPmulambda(c, l)
  n := max(0, - integerBound indicialEq(c, rec.lambda, rec.func))
  zero? e => n::N
  max(n, order(e, c)::Z - rec.mu)::N

hdenom(l, d, e) ==
  */[dd.factor ** NPbound(dd.factor, 1, e)
    for dd in factors balancedFactorisation(d, coefficients l)]

denom0(n, l, d, e, h) ==
  hdenom(l, d, e) * */[hh.factor ** max(0, order(e, hh.factor) - n)::N
    for hh in factors balancedFactorisation(h, e)]
-- returns a polynomials whose zeros are the zeros of e which are not
-- zeros of d
separateZeros(d, e) ==
  ((g := squareFreePart e) exquo gcd(g, squareFreePart d))::UP

indeq(c, l) ==
  rec := NPmulambda(c, l)
  indicialEq(c, rec.lambda, rec.func)

indicialEquations(op:L, p:UP) ==
  [[dd.factor, indeq(dd.factor, op)]
   for dd in factors balancedFactorisation(p, coefficients op)]

-- cannot return "failed" in the homogeneous case

-}
In-field solution of Riccati equations, primitive case.

See Also:
o  )show PrimitiveRatRicDE
package ODEPRRIC PrimitiveRatRicDE

)abbrev package ODEPRRIC PrimitiveRatRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 2 February 1993
++ Description:
++ In-field solution of Riccati equations, primitive case.

PrimitiveRatRicDE(F, UP, L, LQ): Exports == Implementation where
F : Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
UP : UnivariatePolynomialCategory F
L : LinearOrdinaryDifferentialOperatorCategory UP
LQ : LinearOrdinaryDifferentialOperatorCategory Fraction UP

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
UP2 ==> SparseUnivariatePolynomial UP
REC ==> Record(deg:N, eq:UP)
REC2 ==> Record(deg:N, eq:UP2)
POL ==> Record(poly:UP, eq:L)
FRC ==> Record(frac:RF, eq:L)
CNT ==> Record(constant:F, eq:L)
IJ ==> Record(ij: List Z, deg:N)

Exports == with
denomRicDE: L -> UP
++ denomRicDE(op) returns a polynomial \spad{d} such that any rational
++ solution of the associated Riccati equation of \spad{op y = 0} is
++ of the form \spad{p/d + q'/q + r} for some polynomials p and q
++ and a reduced r. Also, \spad{deg(p) < deg(d)} and \{gcd(d,q) = 1\}.

leadingCoefficientRicDE: L -> List REC
++ leadingCoefficientRicDE(op) returns
++ \spad{[[m1, p1], [m2, p2], ... , [mk, pk]]} such that the polynomial
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y = 0} must have degree mj for some j, and its leading
++ coefficient is then a zero of pj. In addition,\spad{m1>m2> ... >mk}.

constantCoefficientRicDE: (L, UP -> List F) -> List CNT
++ constantCoefficientRicDE(op, ric) returns
++ \spad{[[a1, L1], [a2, L2], ... , [ak, Lk]]} such that any rational
++ solution with no polynomial part of the associated Riccati equation of
++ \spad{op y = 0} must be one of the ai's in which case the equation for
++ \spad{z = y e^{-\int ai}} is \spad{Li z = 0}.
++ \spad{ric} is a Riccati equation solver over \spad{F}, whose input
++ is the associated linear equation.

polyRicDE: (L, UP -> List F) -> List POL
++ polyRicDE(op, zeros) returns
++ \spad{[[p1, L1], [p2, L2], ... , [pk, Lk]]} such that the polynomial
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y=0} must be one of the pi's (up to the constant coefficient),
++ in which case the equation for \spad{z=y e^{-\int p}} is \spad{Li z =0}.
++ \spad{zeros} is a zero finder in \spad{UP}.

singRicDE: (L, (UP, UP2) -> List UP, UP -> Factored UP) -> List FRC
++ singRicDE(op, zeros, efactor) returns
++ \spad{[[f1, L1], [f2, L2], \ldots, [fk, Lk]]} such that the singular
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y=0} must be one of the fi's (up to the constant coefficient),
++ in which case the equation for \spad{z=y e^{-\int p}} is \spad{Li z=0}.
++ \spad{zeros(C(x),H(x,y))} returns all the \spad{P_i(x)}'s such that
++ \spad{H(x,P_i(x)) = 0 \text{ modulo } C(x)}.
++ Argument \spad{efactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.

changeVar: (L, UP) -> L
++ changeVar(+/[ai D^i], a) returns the operator \spad{+/[ai (D+a)^i]}.

Implementation ==> add
import PrimitiveRatDE(F, UP, L, LQ)
import BalancedFactorisation(F, UP)

bound : (UP, L) -> N
lambda : (UP, L) -> List IJ
infmax : (IJ, L) -> List Z
dmax : (IJ, UP, L) -> List Z
getPoly : (IJ, UP, L, List Z) -> UP
getPol : (IJ, UP, L, List Z) -> UP2
innerlb : (L, UP -> Z) -> List IJ
innermax : (IJ, L, UP -> Z) -> List Z
tau0 : (UP, UP) -> UP
poly1 : (UP, UP, Z) -> UP2
getPol1 : (List Z, UP, L) -> UP2
getIndices : (N, List IJ) -> List Z
refine : (List UP, UP -> Factored UP) -> List UP
polysol : (L, N, Boolean, UP -> List F) -> List POL
fracsol : (L, (UP, UP2) -> List UP, List UP) -> List FRC
padicsol : (UP, L, N, Boolean, (UP, UP2) -> List UP) -> List FRC
leadingDenomRicDE : (UP, L) -> List REC2
factoredDenomRicDE: L -> List UP
constantCoefficientOperator: (L, N) -> UP
inflambda: L -> List IJ
-- inflambda(op) returns
-- \spad{[[i,j], \deg(a_i)-\deg(a_j)/(i-j)]]} for all the pairs
-- of indices \spad{[i,j]} such that \spad{\deg(a_i)-\deg(a_j)/(i-j)} is
-- an integer.

diff := D()$L
diffq := D()$LQ
\begin{verbatim}
lambda(c, l) == innerlb(l, z +-> order(z, c)::Z)
infLambda l == innerlb(l, z +-> -(degree(z)::Z))
dmax(rec, c, l) == innermax(rec, l, z +-> order(z, c)::Z)
tau0(p, q) == ((q exquo (p ** order(q, p)))::UP) rem p
poly1(c, cp, i) == */[monomial(1,1)$UP2 - (j * cp)::UP2 for j in 0..i-1]
getIndices(n, l) == removeDuplicates_! concat [r.ij for r in l | r.deg=n]
denomRicDE l == */[c ** bound(c, l) for c in factoredDenomRicDE l]
polyRicDE(l, zeros) == concat([0, l], polysol(l, 0, false, zeros))

-- refine([p1,...,pn], foo) refines the list of factors using foo
refine(l, ezfactor) ==
    concat [[r.factor for r in factors ezfactor p] for p in l]

-- returns [] if the solutions of l have no p-adic component at c
padicsol(c, op, b, finite?, zeros) ==
    ans:List(FRC) := empty()
    finite? and zero? b => ans
    lc := leadingDenomRicDE(c, op)
    if finite? then lc := select_!(z +-> z.deg <= b, lc)
    for rec in lc repeat
        for r in zeros(c, rec.eq) | r ^= 0 repeat
            rcn := r /$RF (c ** rec.deg)
            neweq := changeVar(op, rcn)
            sols := padicsol(c, neweq, (rec.deg-1)::N, true, zeros)
            ans :=
                empty? sols => concat([rcn, neweq], ans)
                concat_!([[rcn + sol.frac, sol.eq] for sol in sols], ans)
    ans

leadingDenomRicDE(c, l) ==
    ind:List(Z) -- to cure the compiler... (won't compile without)
    lb := lambda(c, l)
    done:List(N) := empty()
    ans:List(REC2) := empty()
    for rec in lb | (not member?(rec.deg, done)) and
        not(empty?(ind := dmax(rec, c, l))) repeat
        ans := concat([rec.deg, getPol(rec, c, l, ind)], ans)
    done := concat(rec.deg, done)
    sort_!((z1,z2) +-> z1.deg > z2.deg, ans)

getPol(rec, c, l, ind) ==
    -- one?(rec.deg) => getPol1(ind, c, l)
    (rec.deg = 1) => getPol1(ind, c, l)
    +/[monomial(tau0(c, coefficient(l, i::N)), i::N)$UP2 for i in ind]

getPol1(ind, c, l) ==
    cp := diff c
    +/[tau0(c, coefficient(l, i::N)) * poly1(c, cp, i) for i in ind]
\end{verbatim}
constantCoefficientRicDE(op, ric) ==
   m := "max"/[degree p for p in coefficients op]
   [[a, changeVar(op, a::UP)] for a in ric constantCoefficientOperator(op, m)]

constantCoefficientOperator(op, m) ==
   ans:UP := 0
   while op ^= 0 repeat
      if degree(p := leadingCoefficient op) = m then
         ans := ans + monomial(leadingCoefficient p, degree op)
      op := reductum op
   ans

getPoly(rec, l, ind) ==
   +/[monomial(leadingCoefficient coefficient(l,i::N),i::N)$UP for i in ind]
   -- returns empty() if rec is does not reach the max,
   -- the list of indices (including rec) that reach the max otherwise

innermax(rec, l, nu) ==
   n := degree l
   i := first(rec.ij)
   m := i * (d := rec.deg) + nu coefficient(l, i::N)
   ans:List(Z) := empty()
   for j in 0..n | (f := coefficient(l, j)) ^= 0 repeat
      if ((k := (j * d + nu f)) > m) then return empty()
      else if (k = m) then ans := concat(j, ans)
   ans

leadingCoefficientRicDE l ==
   ind:List(Z) -- to cure the compiler... (won't compile without)
   lb := infLambda l
   done:List(N) := empty()
   ans:List(REC) := empty()
   for rec in lb | (not member?(rec.deg, done)) and
      not(empty?(ind := infmax(rec, l))) repeat
      ans := concat([rec.deg, getPoly(rec, l, ind)], ans)
   done := concat(rec.deg, done)
   sort_!((z1,z2) +-> z1.deg > z2.deg, ans)

factoredDenomRicDE l ==
   bd := factors balancedFactorisation(leadingCoefficient l, coefficients l)
   [dd.factor for dd in bd]

changeVar(l:L, a:UP) ==
   dpa := diff + a::L -- the operator (D + a)
   dpan:L := 1 -- will accumulate the powers of (D + a)
   op:L := 0
   for i in 0..degree l repeat
      op := op + coefficient(l, i) * dpan
      dpan := dpa * dpan
primitivePart op

changeVar(l:L, a:RF) ==
  dpa := diffq + a::LQ -- the operator (D + a)
  dpan:LQ := 1 -- will accumulate the powers of (D + a)
  op:LQ := 0
  for i in 0..degree l repeat
    op := op + coefficient(l, i)::RF * dpan
    dpan := dpa * dpan
  splitDenominator(op, empty()).eq

bound(c, l) ==
  empty?(lb := lambda(c, l)) => 1
  "max"/[rec.deg for rec in lb]

innerlb(l, nu) ==
  lb:List(IJ) := empty()
  n := degree l
  for i in 0..n | (li := coefficient(l, i)) ^= 0 repeat
    for j in i+1..n | (lj := coefficient(l, j)) ^= 0 repeat
      u := (nu li - nu lj) exquo (i-j)
      if (u case Z) and ((b := u::Z) > 0) then
        lb := concat([[i, j], b::N], lb)
  lb

singRicDE(l, zeros, ezfactor) ==
  concat([0, l], fracsol(l, zeros, refine(factoredDenomRicDE l, ezfactor)))

fracsol(l, zeros, lc) ==
  ans:List(FRC) := empty()
  empty? lc => ans
  empty? (sols := padicsol(first lc, l, 0, false, zeros)) =>
    fracsol(l, zeros, rest lc)
  for rec in sols repeat
    neweq := changeVar(l, rec.frac)
    sols := fracsol(neweq, zeros, rest lc)
    ans :=
      empty? sols => concat(rec, ans)
      concat_!([ [rec.frac + sol.frac, sol.eq] for sol in sols ]), ans)
    ans

polysol(l, b, finite?, zeros) ==
  ans:List(POL) := empty()
  finite? and zero? b => ans
  lc := leadingCoefficientRicDE l
  if finite? then lc := select_!(z -> z.deg <= b, lc)
for rec in lc repeat
  for a in zeros(rec.eq) | a ^= 0 repeat
    atn UP := monomial(a, rec.deg)
    neweq := changeVar(l, atn)
    sols := polysol(neweq, (rec.deg - 1)::N, true, zeros)
    ans :=
      empty? sols => concat([atn, neweq], ans)
      concat_!([[atn + sol.poly, sol.eq] for sol in sols], ans)
    ans

package PRINT PrintPackage

— PrintPackage.input —

)set break resume
)sys rm -f PrintPackage.output
)spool PrintPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrintPackage
--E 1

)spool
)lisp (bye)

— PrintPackage.help —

====================================================================
PrintPackage examples
=====================================================================
PrintPackage provides a print function for output forms.

See Also:
  o )show PrintPackage

PrintPackage (PRINT)

Exports:
print

— package PRINT PrintPackage —

)abbrev package PRINT PrintPackage
++ Author: Scott Morrison
++ Date Created: Aug. 1, 1990
++ Description:
  + PrintPackage provides a print function for output forms.

PrintPackage(): with
  print : OutputForm -> Void
    ++ print(o) writes the output form o on standard output using the
    ++ two-dimensional formatter.
  == add
    print(x) == print(x)$OutputForm
package PSEUDLIN PseudoLinearNormalForm

PseudoLinearNormalForm examples

PseudoLinearNormalForm provides a function for computing a block-companion form for pseudo-linear operators.

See Also:
- )show PseudoLinearNormalForm
PseudoLinearNormalForm (PSEUDLIN)

Exports:
changeBase  companionBlocks  normalForm

package PSEUDLIN PseudoLinearNormalForm

PseudoLinearNormalForm(K:Field): Exports == Implementation where
ER ==> Record(C: Matrix K, g: Vector K)
REC ==> Record(R: Matrix K, A: Matrix K, Ainv: Matrix K)

Exports == with
normalForm: (Matrix K, Automorphism K, K -> K) -> REC
  ++ normalForm(M, sig, der) returns \spad{[R, A, A^{-1}]} such that
  ++ the pseudo-linear operator whose matrix in the basis \spad{y} is
  ++ \spad{M} had matrix \spad{R} in the basis \spad{z = A y}.
  ++ \spad{der} is a \spad{sig}-derivation.
changeBase: (Matrix K, Matrix K, Automorphism K, K -> K) -> Matrix K
  ++ changeBase(M, A, sig, der): computes the new matrix of a pseudo-linear
  ++ transform given by the matrix \spad{M} under the change of base \spad{A}
companionBlocks: (Matrix K, Vector K) -> List ER
  ++ companionBlocks(m, v) returns \spad{[[C_1, g_1],...,C_k, g_k]]}
  ++ such that each \spad{[C_i]} is a companion block and
  ++ \spad{m = diagonal(C_1,...,C_k)}.

Implementation == add
normalForm0: (Matrix K, Automorphism K, Automorphism K, K -> K) -> REC
mulMatrix: (Integer, Integer, K) -> Matrix K
  -- mulMatrix(N, i, a): under a change of base with the resulting matrix of
-- size N*N the following operations are performed:
-- D1: column i will be multiplied by sig(a)
-- D2: row i will be multiplied by 1/a
-- D3: addition of der(a)/a to the element at position (i,i)
addMatrix: (Integer, Integer, Integer, K) \rightarrow Matrix K

-- addMatrix(N, i, k, a): under a change of base with the resulting matrix
-- of size N*N the following operations are performed:
-- C1: addition of column i multiplied by sig(a) to column k
-- C2: addition of row k multiplied by -a to row i
-- C3: addition of -a*der(a) to the element at position (i,k)
permutationMatrix: (Integer, Integer, Integer) \rightarrow Matrix K

-- permutationMatrix(N, i, k): under a change of base with the resulting
-- permutation matrix of size N*N the following operations are performed:
-- P1: columns i and k will be exchanged
-- P2: rows i and k will be exchanged
inv: Matrix K \rightarrow Matrix K

-- inv(M): computes the inverse of a invertable matrix M.
-- avoids possible type conflicts

inv m == inverse(m) :: Matrix K

cchangeBase(M, A, sig, der) ==
  inv(A) * (M * map((k1:K):K +-> sig k1, A) + map(der, A))

normalForm(M, sig, der) == normalForm0(M, sig, inv sig, der)

companionBlocks(R, w) ==

-- decomposes the rational matrix R into single companion blocks
-- and the inhomogenity w as well
i:Integer := 1
n := nrows R
l:List(ER) := empty()
while i <= n repeat
  j := i
  while j+1 <= n and R(j,j+1) = 1 repeat j := j+1
  --split block now
  v:Vector K := new((j-i+1)::NonNegativeInteger, 0)
  for k in i..j repeat v(k-i+1) := w k
  l := concat([subMatrix(R,i,j,i,j), v], l)
i := j+1
l

normalForm0(M, sig, siginv, der) ==

-- the changes of base will be incremented in B and Binv,
-- where B**(-1)=Binv; E defines an elementary matrix
B, Binv, E : Matrix K
recOfMatrices : REC
N := nrows M
B := diagonalMatrix [1 for k in 1..N]
Binv := copy B

-- avoid unnecessary recursion
if diagonal?(M) then return [M, B, Binv]
i := Integer := 1
while i < N repeat
  j := i + 1
  while j <= N and M(i, j) = 0 repeat j := j + 1
  if j <= N then
    -- expand companionblock by lemma 5
    if j ≠ i+1 then
      -- perform first a permutation
      E := permutationMatrix(N, i+1, j)
      M := changeBase(M, E, sig, der)
      B := B*E
      Binv := E*Binv
    -- now is M(i, i+1) ≠ 0
    E := mulMatrix(N, i+1, siginv inv M(i,i+1))
    M := changeBase(M, E, sig, der)
    B := B*E
    Binv := inv(E)*Binv
    for j in 1..N repeat
      if j ≠ i+1 then
        E := addMatrix(N, i+1, j, siginv(-M(i,j)))
        M := changeBase(M, E, sig, der)
        B := B*E
        Binv := inv(E)*Binv
      i := i + 1
    else
      -- apply lemma 6
      for j in i..2 by -1 repeat
        for k in (i+1)..N repeat
          E := addMatrix(N, k, j-1, M(k,j))
          M := changeBase(M, E, sig, der)
          B := B*E
          Binv := inv(E)*Binv
        j := i + 1
      while j <= N and M(j,1) = 0 repeat j := j + 1
    if j <= N then
      -- expand companionblock by lemma 8
      E := permutationMatrix(N, 1, j)
      M := changeBase(M, E, sig, der)
      B := B*E
      Binv := E*Binv
    -- start again to establish rational form
    i := 1
  else
    -- split a direct factor
    recOfMatrices :=
      normalForm(subMatrix(M, i+1, N, i+1, N), sig, der)
    setsubMatrix!(M, i+1, i+1, recOfMatrices.R)
    E := diagonalMatrix [1 for k in 1..N]
    setsubMatrix!(E, i+1, i+1, recOfMatrices.A)
    B := B*E
setsubMatrix!(E, i+1, i+1, recOfMatrices.Ainv)
Binv := E*Binv
-- M in blockdiagonalform, stop program
i := N
[M, B, Binv]

mulMatrix(N, i, a) ==
M : Matrix K := diagonalMatrix [1 for j in 1..N]
M(i, i) := a
M

addMatrix(N, i, k, a) ==
A : Matrix K := diagonalMatrix [1 for j in 1..N]
A(i, k) := a
A

permutationMatrix(N, i, k) ==
P : Matrix K := diagonalMatrix [1 for j in 1..N]
P(i, i) := P(k, k) := 0
P(i, k) := P(k, i) := 1
P

package PRS PseudoRemainderSequence

--PSEUDLIN.dotabb--

"PSEUDLIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSEUDLIN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"PSEUDLIN" -> "IVECTOR"
The package constructor PseudoRemainderSequence provides efficient algorithms by Lionel Ducos (University of Poitiers, France) for computing sub-resultants. This leads to a speed up in many places in Axiom where sub-resultants are computed (polynomial system solving, algebraic factorization, integration).

This package contains some functions from improvements of the subresultants algorithm.

See Also:
  o )show PseudoRemainderSequence
--- package PRS PseudoRemainderSequence ---

\begin{verbatim}
package PRS PseudoRemainderSequence
++ Author: Ducos Lionel (Lionel.Ducos@mathlabo.univ-poitiers.fr)
++ Date Created: january 1995
++ Date Last Updated: 5 february 1999
++ References:
++ Lionel Ducos ‘‘Optimizations of the subresultant algorithm’’
++ Description:
++ This package contains some functions: discriminant, resultant,
++ subresultantGcd, chainSubresultants, degreeSubresultant, lastSubresultant,
++ resultantEuclidean, subresultantGcdEuclidean, semiresultantGcdEuclidean1,
++ semiresultantGcdEuclidean2
++ These procedures come from improvements of the subresultants algorithm.

PseudoRemainderSequence(R, polR) : Specification == Implementation where
  R : IntegralDomain
  polR : UnivariatePolynomialCategory(R)
  NNI ==> NonNegativeInteger
  LC ==> leadingCoefficient

Specification == with
  resultant : (polR, polR) -> R
    ++ \texttt{resultant}(P, Q) returns the resultant
    ++ of \texttt{resultant}(P) and \texttt{resultant}(Q)

  resultantEuclidean : (polR, polR) ->
    Record(coef1 : polR, coef2 : polR, resultant : R)
    ++ \texttt{resultantEuclidean}(P, Q) carries out the equality
\end{verbatim}
++ \[\text{axiom}\{\text{coef1}\cdot P + \text{coef2}\cdot Q = \text{resultant}(P,Q)\}\]

semiResultantEuclidean2 : (polR, polR) ->
  Record(coef2 : polR, resultant : R)
  ++ \[\text{axiom}\{\text{semiResultantEuclidean2}(P,Q)\} \text{ carries out the equality} \]
  ++ \[\text{axiom}\{\ldots P + \text{coef2}\cdot Q = \text{resultant}(P,Q)\}\].
  ++ \text{Warning.} \[\text{axiom}\{\text{degree}(P) \geq \text{degree}(Q)\}\].

semiResultantEuclidean1 : (polR, polR) ->
  Record(coef1 : polR, resultant : R)
  ++ \[\text{axiom}\{\text{semiResultantEuclidean1}(P,Q)\} \text{ carries out the equality} \]
  ++ \[\text{axiom}\{\ldots P + ? Q = \text{resultant}(P,Q)\}\].

indiceSubResultant : (polR, polR, NNI) -> polR
  ++ \[\text{axiom}\{\text{indiceSubResultant}(P, Q, i)\} \text{ returns} \]
  ++ \text{the subresultant of indice} \[\text{axiom}\{i\}\]

indiceSubResultantEuclidean : (polR, polR, NNI) ->
  Record(coef1 : polR, coef2 : polR, subResultant : polR)
  ++ \[\text{axiom}\{\text{indiceSubResultantEuclidean}(P, Q, i)\} \text{ returns} \]
  ++ \text{the subresultant} \[\text{axiom}\{S_i(P,Q)\}\] \text{and carries out the equality} \]
  ++ \[\text{axiom}\{\ldots P + \text{coef2}\cdot Q = S_i(P,Q)\}\]
  ++ \text{Warning.} \[\text{axiom}\{\text{degree}(P) \geq \text{degree}(Q)\}\].

semiIndiceSubResultantEuclidean : (polR, polR, NNI) ->
  Record(coef2 : polR, subResultant : polR)
  ++ \[\text{axiom}\{\text{semiIndiceSubResultantEuclidean}(P, Q, i)\} \text{ returns} \]
  ++ \text{the subresultant} \[\text{axiom}\{S_i(P,Q)\}\] \text{and carries out the equality} \]
  ++ \[\text{axiom}\{\ldots P + \text{coef2}\cdot Q = S_i(P,Q)\}\]
  ++ \text{Warning.} \[\text{axiom}\{\text{degree}(P) \geq \text{degree}(Q)\}\].

degreeSubResultant : (polR, polR, NNI) -> polR
  ++ \[\text{axiom}\{\text{degreeSubResultant}(P, Q, d)\} \text{ computes} \]
  ++ \text{a subresultant of degree} \[\text{axiom}\{d\}\].

degreeSubResultantEuclidean : (polR, polR, NNI) ->
  Record(coef1 : polR, coef2 : polR, subResultant : polR)
  ++ \[\text{axiom}\{\text{degreeSubResultantEuclidean}(P, Q, i)\} \text{ returns} \]
  ++ \text{a subresultant} \[\text{axiom}\{S\}\] \text{of degree} \[\text{axiom}\{d\}\]
  ++ \text{and carries out the equality} \[\text{axiom}\{\text{coef1}\cdot P + \text{coef2}\cdot Q = S_i\}\].

semiDegreeSubResultantEuclidean : (polR, polR, NNI) ->
  Record(coef2 : polR, subResultant : polR)
  ++ \[\text{axiom}\{\text{semiDegreeSubResultantEuclidean}(P, Q, i)\} \text{ returns} \]
  ++ \text{a subresultant} \[\text{axiom}\{S\}\] \text{of degree} \[\text{axiom}\{d\}\]
  ++ \text{and carries out the equality} \[\text{axiom}\{\ldots P + \text{coef2}\cdot Q = S_i\}\].
  ++ \text{Warning.} \[\text{axiom}\{\text{degree}(P) \geq \text{degree}(Q)\}\].

lastSubResultant : (polR, polR) -> polR
  ++ \[\text{axiom}\{\text{lastSubResultant}(P, Q)\} \text{ computes} \]
  ++ \text{the last non zero subresultant of} \[\text{axiom}\{P\}\] \text{and} \[\text{axiom}\{Q\}\]
lastSubResultantEuclidean : (polR, polR) ->
   Record(coef1 : polR, coef2 : polR, subResultant : polR)
   ++ \texttt{lastSubResultantEuclidean(P, Q)} computes
   ++ the last non zero subresultant \texttt{S}
   ++ and carries out the equality \texttt{coef1*P + coef2*Q = S}.

semiLastSubResultantEuclidean : (polR, polR) ->
   Record(coef2 : polR, subResultant : polR)
   ++ \texttt{semiLastSubResultantEuclidean(P, Q)} computes
   ++ the last non zero subresultant \texttt{S}
   ++ and carries out the equality \texttt{...P + coef2*Q = S}.
   ++ Warning. \texttt{degree(P) >= degree(Q)}.

subResultantGcd : (polR, polR) -> polR
   ++ \texttt{subResultantGcd(P, Q)} returns the gcd
   ++ of two primitive polynomials \texttt{P} and \texttt{Q}.

subResultantGcdEuclidean : (polR, polR) -> Record(coef1 : polR, coef2 : polR, gcd : polR)
   ++ \texttt{subResultantGcdEuclidean(P,Q)} carries out the equality
   ++ \texttt{coef1*P + coef2*Q = +/- S_i(P,Q)}
   ++ where the degree (not the indice)
   ++ of the subresultant \texttt{S_i(P,Q)} is the smaller as possible.

semiSubResultantGcdEuclidean2 : (polR, polR) -> Record(coef2 : polR, gcd : polR)
   ++ \texttt{semiSubResultantGcdEuclidean2(P,Q)} carries out the equality
   ++ \texttt{...P + coef2*Q = +/- S_i(P,Q)}
   ++ where the degree (not the indice)
   ++ of the subresultant \texttt{S_i(P,Q)} is the smaller as possible.
   ++ Warning. \texttt{degree(P) >= degree(Q)}.

semiSubResultantGcdEuclidean1: (polR, polR) -> Record(coef1 : polR, gcd : polR)
   ++ \texttt{semiSubResultantGcdEuclidean1(P,Q)} carries out the equality
   ++ \texttt{coef1*P + ? Q = +/- S_i(P,Q)}
   ++ where the degree (not the indice)
   ++ of the subresultant \texttt{S_i(P,Q)} is the smaller as possible.

discriminant : polR -> R
   ++ \texttt{discriminant(P, Q)} returns the discriminant
   ++ of \texttt{P} and \texttt{Q}.

discriminantEuclidean : polR -> Record(coef1 : polR, coef2 : polR, discriminant : R)
   ++ \texttt{discriminantEuclidean(P)} carries out the equality
   ++ \texttt{coef1 * P + coef2 * D(P) = discriminant(P)}.

semiDiscriminantEuclidean : polR ->
   Record(coef2 : polR, discriminant : R)
++ \texttt{discriminantEuclidean(P)} carries out the equality
++ \texttt{P + coef2 * D(P) = discriminant(P)}.
++ Warning. \texttt{\text{degree}(P) \geq \text{degree}(Q)}.

\textbf{chainSubResultants : (polR, polR) \rightarrow List(polR)}
++ \texttt{\text{chainSubResultants}(P, Q)} computes the list
++ of non zero subresultants of \texttt{axiom(P)} and \texttt{axiom(Q)}.

\textbf{schema : (polR, polR) \rightarrow List(NNI)}
++ \texttt{\text{schema}(P,Q)} returns the list of degrees of
++ non zero subresultants of \texttt{axiom(P)} and \texttt{axiom(Q)}.

if \texttt{R} has \texttt{GcdDomain} then
\textbf{resultantReducit : (polR, polR) \rightarrow R}
++ \texttt{\text{resultantReducit}(P,Q)} returns the "reduce resultant"
++ of \texttt{\text{axiom}(P)} and \texttt{\text{axiom}(Q)}.

\textbf{resultantReducitEuclidean : (polR, polR) \rightarrow}
++ \texttt{Record(coef1 : polR, coef2 : polR, resultantReducit : R)}
++ the "reduce resultant" and carries out the equality
++ \texttt{\text{coef1}\text{P} + \text{coef2}\text{Q} = \text{resultantReducit}(P,Q)}.

\textbf{semiResultantReducitEuclidean : (polR, polR) \rightarrow}
++ \texttt{Record(coef2 : polR, resultantReducit : R)}
++ the "reduce resultant" and carries out the equality
++ \texttt{\text{...P} + \text{coef2}\text{Q} = \text{resultantReducit}(P,Q)}.

\textbf{gcd : (polR, polR) \rightarrow polR}
++ \texttt{\text{gcd}(P, Q)} returns the gcd of \texttt{\text{axiom}(P)} and \texttt{\text{axiom}(Q)}.

\textbf{-- sub-routines exported for convenience -----------------------------}

\textbf{\texttt{*} : (R, Vector(polR)) \rightarrow Vector(polR)}
++ \texttt{\text{r \ast v}} computes the product of \texttt{\text{r}} and \texttt{\text{v}}

\textbf{\texttt{exquo} : (Vector(polR), R) \rightarrow Vector(polR)}
++ \texttt{\text{v \text{exquo} r}} computes
++ the exact quotient of \texttt{\text{v}} by \texttt{\text{r}}

\textbf{pseudoDivide : (polR, polR) \rightarrow}
++ \texttt{Record(coef:R, quotient:polR, remainder:polR)}
++ \texttt{\text{pseudoDivide}(P,Q)} computes the pseudoDivide
++ of \texttt{\text{axiom}(P)} by \texttt{\text{axiom}(Q)}.

\textbf{divide : (polR, polR) \rightarrow Record(quotient : polR, remainder : polR)}
++ \texttt{\text{divide}(F,G)} computes quotient and rest
++ of the exact euclidean division of \texttt{\text{axiom}(F)} by \texttt{\text{axiom}(G)}.
Lazard : (R, R, NNI) -> R
++ \axiom{Lazard(x, y, n)} computes \axiom{x**n/y**(n-1)}

Lazard2 : (polR, R, R, NNI) -> polR
++ \axiom{Lazard2(F, x, y, n)} computes \axiom{(x/y)**(n-1) * F}

next_sousResultant2 : (polR, polR, polR, R) -> polR
++ \axiom{nextsousResultant2(P, Q, Z, s)} returns
++ the subresultant \axiom{S_{e-1}} where
++ \axiom{P \sim S_d, Q = S_{d-1}, Z = S_e, s = lc(S_d)}

resultant_naif : (polR, polR) -> R
++ \axiom{resultantEuclidean_naif(P,Q)} returns
++ the resultant of \axiom{P} and \axiom{Q} computed
++ by means of the naive algorithm.

resultantEuclidean_naif : (polR, polR) -> Record(coef1 : polR, coef2 : polR, resultant : R)
++ \axiom{resultantEuclidean_naif(P,Q)} returns
++ the extended resultant of \axiom{P} and \axiom{Q} computed
++ by means of the naive algorithm.

semiResultantEuclidean_naif : (polR, polR) -> Record(coef2 : polR, resultant : R)
++ \axiom{resultantEuclidean_naif(P,Q)} returns
++ the semi-extended resultant of \axiom{P} and \axiom{Q} computed
++ by means of the naive algorithm.

Implementation == add
X : polR := monomial(1$R,1)

r : R * v : Vector(polR) == r::polR * v
-- the instruction map(r * #1, v) is slower !?

v : Vector(polR) exquo r : R ==
map((p1:polR):polR +-> (p1 exquo r)::polR, v)

c odeDivide(P : polR, Q : polR) :
Record(coef:R,quotient:polR,remainder:polR) ==
-- computes the pseudoDivide of P by Q
zero?(Q) => error("PseudoDivide$PRS : division by 0")
zero?(P) => construct(1, 0, P)
lcQ : R := LC(Q)
(d egP, degQ) := (degree(P), degree(Q))
degP < degQ => construct(1, 0, P)
Q := reductum(Q)
i : NNI := (degP - degQ + 1)::NNI
co : R := lcQ**i
quot : polR := 0$polR
while (delta : Integer := degree(P) - degQ) >= 0 repeat
i := (i - 1)::NNI
mon := monomial(LC(P), delta::NNI)$polR
quot := quot + lcQ**i * mon
P := lcQ * reductum(P) - mon * Q
P := lcQ**i * P
return construct(co, quot, P)

divide(F : polR, G : polR) : Record(quotient : polR, remainder : polR)==
-- computes quotient and rest of the exact euclidean division of F by G
1cG : R := LC(G)
degG : NNI := degree(G)
zero?(degG) => ( F := (F exquo 1cG)::polR; return construct(F, 0))
G : polR := reductum(G)
quot : polR := 0
while (delta := degree(F) - degG) >= 0 repeat
mon : polR := monomial((LC(F) exquo 1cG)::R, delta::NNI)
quot := quot + mon
F := reductum(F) - mon * G
return construct(quot, F)

resultant_naif(P : polR, Q : polR) : R ==
-- valid over a field
a : R := 1
repeat
zero?(Q) => return 0
(degP, degQ) := (degree(P), degree(Q))
if odd?(degP) and odd?(degQ) then a := -a
zero?(degQ) => return (a * LC(Q)**degP)
U : polR := divide(P, Q).remainder
a := a * LC(Q)**(degP - degree(U))::NNI
(P, Q) := (Q, U)

resultantEuclidean_naif(P : polR, Q : polR) :
Record(coef1 : polR, coef2 : polR, resultant : R) ==
-- valid over a field.
a : R := 1
old_cf1 : polR := 1 ; cf1 : polR := 0
old_cf2 : polR := 0 ; cf2 : polR := 1
repeat
zero?(Q) => construct(0::polR, 0::polR, 0::R)
(degP, degQ) := (degree(P), degree(Q))
if odd?(degP) and odd?(degQ) then a := -a
if zero?(degQ) then
a := a * LC(Q)**(degP-1):NNI
return construct(a*cf1, a*cf2, a*LC(Q))
divid := divide(P,Q)
a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
(P, Q) := (Q, divid.remainder)
(old_cf1, old_cf2, cf1, cf2) := (cf1, cf2,
old_cf1 - divid.quotient * cf1, old_cf2 - divid.quotient * cf2)
semiResultantEuclidean_naif(P : polR, Q : polR) :
  Record(coef2 : polR, resultant : R) ==
  -- valid over a field
  a : R := 1
  old_coef2 : polR := 0 ; cf2 : polR := 1
  repeat
    zero?(Q) => construct(0::polR, 0::R)
    (degP, degQ) := (degree(P), degree(Q))
    if odd?(degP) and odd?(degQ) then a := -a
    if zero?(degQ) then
      a := a * LC(Q)**(degP-1)::NNI
      return construct(a*cf2, a*LC(Q))
    divid := divide(P,Q)
    a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
    (P, Q) := (Q, divid.remainder)
    (old_coef2, cf2) := (cf2, old_coef2 - divid.quotient * cf2)

Lazard(x : R, y : R, n : NNI) : R ==
  zero?(n) => error("Lazard$PRS : n = 0")
  -- one?(n) => x
  (n = 1) => x
  a : NNI := 1
  while n >= (b := 2*a) repeat a := b
  c : R := x
  n := (n - a)::NNI
  repeat -- c = x**i / y**(i-1), i=n_0 quo a, a=2**?
    -- one?(a) => return c
    (a = 1) => return c
    a := a quo 2
    c := ((c * c) exquo y)::R
    if n >= a then ( c := ((c * x) exquo y)::R ; n := (n - a)::NNI )

Lazard2(F : polR, x : R, y : R, n : NNI) : polR ==
  zero?(n) => error("Lazard2$PRS : n = 0")
  -- one?(n) => F
  (n = 1) => F
  x := Lazard(x, y, (n-1)::NNI)
  return ((x * F) exquo y)::polR

Lazard3(V : Vector(polR), x : R, y : R, n : NNI) : Vector(polR) ==
  -- computes x**(n-1) * V / y**(n-1)
  zero?(n) => error("Lazard2$prs : n = 0")
  -- one?(n) => V
  (n = 1) => V
  x := Lazard(x, y, (n-1)::NNI)
  return ((x * V) exquo y)

  (lcP, c, se) := (LC(P), LC(Q), LC(Z))
(d, e) := (degree(P), degree(Q))
(P, Q, H) := (reductum(P), reductum(Q), - reductum(Z))
A : polR := coefficient(P, e) * H
for i in e+1..d-1 repeat
  H := if degree(H) = e-1 then
       X * reductum(H) - ((LC(H) * Q) exquo c)::polR
  else
       X * H
  A := (A exquo lcP)::polR
A := A + se * P -- A = s_e * reductum(P_0) mod S_d-1
A := if degree(H) = e-1 then
     c * (X * reductum(H) + A) - LC(H) * Q
else
     c * (X * H + A)
A := (A exquo s)::polR -- A = +/- S_e-1
return (if odd?(d-e) then A else - A)

next_sousResultant3(VP : Vector(polR), VQ : Vector(polR), s : R, ss : R) :
  Vector(polR) ==
  -- P ~ S_d, Q = S_d-1, s = lc(P), ss = lc(Q)
  (P, Q) := (VP.1, VQ.1)
  (lcP, c) := (LC(P), LC(Q))
  e : NNI := degree(Q)
  -- if one?(delta := degree(P) - e) then -- algo_new
  if ((delta := degree(P) - e) = 1) then -- algo_new
    VP := c * VP - coefficient(P, e) * VQ
    VP := VP exquo lcP
    VP := c * (VP - X * VQ) + coefficient(Q, (e-1)::NNI) * VQ
    VP := VP exquo s
  else
    -- algorithm of Lickteig - Roy
    (r, rr) := (s * lcP, ss * c)
    divid := divide(rr * P, Q)
    VP.1 := (divid.remainder exquo r)::polR
    for i in 2..#VP repeat
      VP.i := rr * VP.i - VQ.i * divid.quotient
      VP.i := (VP.i exquo r)::polR
    return (if odd?(delta) then VP else - VP)

algo_new(P : polR, Q : polR) : R ==
  delta : NNI := (degree(P) - degree(Q))::NNI
  s : R := LC(Q)**delta
  (P, Q) := (Q, pseudoRemainder(P, -Q))
  repeat
    -- P = S_c-1 (except the first turn : P ~ S_c-1),
    -- Q = S_d-1,  s = lc(S_d)
    zero?(Q) => return 0
    delta := (degree(P) - degree(Q))::NNI
Z := Lazard2(Q, LC(Q), s, delta)
-- Z = S_e ~ S_d-1
zero?(degree(Z)) => return LC(Z)
(P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

resultant(P : polR, Q : polR) : R ==
zero?(Q) or zero?(P) => 0
if degree(P) < degree(Q) then
(P, Q) := (Q, P)
if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
zero?(degree(Q)) => LC(Q)**degree(P)
-- degree(P) >= degree(Q) > 0
R has Finite => resultant_naif(P, Q)
return algo_new(P, Q)

subResultantEuclidean(P : polR, Q : polR) :
Record(coef1 : polR, coef2 : polR, resultant : R) ==
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
-- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
-- S_{c-1} = VP.2 P_0 + VP.3 Q_0, S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
(P, Q) := (VP.1, VQ.1)
zero?(Q) => return construct(0::polR, 0::polR, 0::R)
e : NNI := degree(Q)
delta : NNI := (degree(P) - e)::NNI
if zero?(e) then
  l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
  return construct(1.2, 1.3, LC(1.1))
ss : R := Lazard(LC(Q), s, delta)
(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss
resultantEuclidean(P : polR, Q : polR) :
Record(coef1 : polR, coef2 : polR, resultant : R) ==
zero?(P) or zero?(Q) => construct(0::polR, 0::polR, 0::R)
if degree(P) < degree(Q) then
e : Integer := if odd?(degree(P)) and odd?(degree(Q)) then -1 else 1
l := resultantEuclidean(Q, e * P)
return construct(e * l.coef2, l.coef1, l.resultant)
if zero?(degree(Q)) then
dep : NNI := degree(P)
zero?(degP) => error("resultantEuclidean: constant polynomials")
s : R := LC(Q)**(degP-1)::NNI
return construct(0::polR, s::polR, s * LC(Q))
R has Finite => resultantEuclidean_naif(P, Q)
return subResultantEuclidean(P, Q)
semiSubResultantEuclidean(P : polR, Q : polR) :
    Record(coef2 : polR, resultant : R) ==
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
    repeat
        -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
        -- S_{c-1} = ...P_0 + VP.3 Q_0, S_{d-1} = ...P_0 + VQ.3 Q_0
        (P, Q) := (VP.1, VQ.1)
        zero?(Q) => return construct(0::polR, 0::R)
        e : NNI := degree(Q)
        delta : NNI := (degree(P) - e)::NNI
        if zero?(e) then
            l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
            return construct(l.2, LC(l.1))
        ss : R := Lazard(LC(Q), s, delta)
        (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
        s := ss
    semiResultantEuclidean2(P : polR, Q : polR) :
    Record(coef2 : polR, resultant : R) ==
    zero?(P) or zero?(Q) => construct(0::polR, 0::R)
    degree(P) < degree(Q) => error("semiResultantEuclidean2 : bad degrees")
    if zero?(degree(Q)) then
        degP : NNI := degree(P)
        zero?(degP) => error("semiResultantEuclidean2 : constant polynomials")
        s : R := LC(Q)**(degP-1)::NNI
        return construct(s::polR, s * LC(Q))
    R has Finite => semiResultantEuclidean_naif(P, Q)
    return semiSubResultantEuclidean(P,Q)
semiResultantEuclidean1(P : polR, Q : polR) :
    Record(coef1 : polR, resultant : R) ==
    result := resultantEuclidean(P,Q)
    [result.coef1, result.resultant]
indiceSubResultant(P : polR, Q : polR, i : NNI) : polR ==
    zero?(Q) or zero?(P) => 0
    if degree(P) < degree(Q) then
        (P, Q) := (Q, P)
        if odd?(degree(P)-i) and odd?(degree(Q)-i) then Q := - Q
    if i = degree(Q) then
        delta : NNI := (degree(P)-degree(Q))::NNI
        zero?(delta) => error("indiceSubResultant$PRS : bad degrees")
        s : R := LC(Q)**(delta-1)::NNI
        return s*Q
    i > degree(Q) => 0
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
  -- P = S_{c-1} \cdot S_d , Q = S_{d-1} , s = lc(S_d) , i < d
  if (degP, degQ) := (degree(P), degree(Q))
  i = degP-1 => return Q
  zero?(Q) or (i > degQ) => return 0
  Z : polR := Lazard2(Q, LC(Q), s, (degP - degQ)::NNI)
  -- Z = S_e \cdot S_{d-1}
  i = degQ => return Z
  (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
  s := LC(Z)

indiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
  Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
  if degree(P) < degree(Q) then
    e := if odd?(degree(P)-i) and odd?(degree(Q)-i) then -1 else 1
    l := indiceSubResultantEuclidean(Q, e * P, i)
    return construct(e * l.coef2, l.coef1, l.subResultant)
  if i = degree(Q) then
    delta : NNI := (degree(P)-degree(Q))::NNI
    zero?(delta) =>
      error("indiceSubResultantEuclidean$PRS : bad degrees")
    s : R := LC(Q)**(delta-1)::NNI
    return construct(0::polR, s::polR, s * Q)
  i > degree(Q) => construct(0::polR, 0::polR, 0::polR)
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  VP : Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
  VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
  -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d), i < d
  -- S_{c-1} = VP.2 P_0 + VP.3 Q_0 , S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
  (P, Q) := (VP.1, VQ.1)
  zero?(Q) => return construct(0::polR, 0::polR, 0::polR)
  if (degP, degQ) := (degree(P), degree(Q))
  i > degQ => return construct(0::polR, 0::polR, 0::polR)
  VZ := Lazard3(VQ, LC(Q), s, (degP - degQ)::NNI)
  i = degQ => return construct(VZ.2, VZ.3, VZ.1)
  ss : R := LC(VZ.1)
  (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, ss))
s := ss

semiIndiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
  Record(coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
  if degree(P) < degree(Q) then
    error("semiIndiceSubResultantEuclidean$PRS : bad degrees")

delta : NNI := (degree(P) - degree(Q))::NNI
zero?(delta) =>
    error("semiIndiceSubResultantEuclidean$PRS : bad degrees")
s : R := LC(Q)**(delta-1)::NNI
return construct(s::polR, s * Q)
i > degree(Q) => construct(0::polR, 0::polR)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = lc(S_d), i < d
    -- S_{c-1} = ...P_0 + VP.2 Q_0, S_{d-1} = ...P_0 + ...Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) or (i > degree(Q)) => return 0
    i = degree(Q) => return Q
    Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
    -- Z = S_e ~ S_d-1
    (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

degreeSubResultant(P : polR, Q : polR, i : NNI) : polR ==
zero?(Q) or zero?(P) => 0
if degree(P) < degree(Q) then (P, Q) := (Q, P)
if i = degree(Q) then
    delta : NNI := (degree(P) - degree(Q))::NNI
    zero?(delta) => error("degreeSubResultant$PRS : bad degrees")
s : R := LC(Q)**(delta-1)::NNI
    return s*Q
    i > degree(Q) => 0
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
    -- P = S_{c-1}, Q = S_{d-1}, s = lc(S_d)
    zero?(Q) or (i > degree(Q)) => return 0
    i = degree(Q) => return Q
    Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
    -- Z = S_e ~ S_d-1
    (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

degreeSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
    Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
if degree(P) < degree(Q) then
    l := degreeSubResultantEuclidean(Q, P, i)
    return construct(l.coef2, l.coef1, l.subResultant)
if i = degree(Q) then
    delta := \((\text{degree}(P) - \text{degree}(Q))\)::NNI
    zero?(delta) =>
        error("\text{degreeSubResultantEuclideanPRS : bad degrees}\)
    s := \text{LC}(Q)**(\text{delta-1})::NNI
    return construct(0::polR, s::polR, s * Q)

i > degree(Q) => construct(0::polR, 0::polR, 0::polR)

s := \text{LC}(Q)**(\text{degree}(P) - \text{degree}(Q))::NNI
VP := Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ := Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = \text{lc}(S_d)
    -- S_{c-1} = \ldots P_0 + VP.3 Q_0, S_{d-1} = \ldots P_0 + VQ.3 Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) or (i > degree(Q)) =>
        return construct(0::polR, 0::polR, 0::polR)
    i = degree(Q) => return construct(VQ.2, VQ.3, VQ.1)
    ss := \text{Lazard}(\text{LC}(Q), s, (\text{degree}(P) - \text{degree}(Q))::NNI)
    (VP, VQ) := (VQ, \text{next_sousResultant3}(VP, VQ, s, ss))
    s := ss

semiDegreeSubResultantEuclidean(P : polR, Q : polR, i : NNI) :=
    Record(coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
     degree(P) < degree(Q) =>
        error("\text{semiDegreeSubResultantEuclideanPRS : bad degrees}\)
    if i = degree(Q) then
        delta := \((\text{degree}(P) - \text{degree}(Q))\)::NNI
        zero?(delta) =>
            error("\text{semiDegreeSubResultantEuclideanPRS : bad degrees}\)
        s := \text{LC}(Q)**(\text{delta-1})::NNI
        return construct(s::polR, s * Q)
    i > degree(Q) => construct(0::polR, 0::polR)
    s := \text{LC}(Q)**(\text{degree}(P) - \text{degree}(Q))::NNI
    VP := Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ := Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = \text{lc}(S_d)
    -- S_{c-1} = \ldots P_0 + VP.3 Q_0, S_{d-1} = \ldots P_0 + VQ.3 Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) or (i > degree(Q)) =>
        return construct(0::polR, 0::polR)
    i = degree(Q) => return construct(VQ.2, VQ.3, VQ.1)
    ss := \text{Lazard}(\text{LC}(Q), s, (\text{degree}(P) - \text{degree}(Q))::NNI)
    (VP, VQ) := (VQ, \text{next_sousResultant3}(VP, VQ, s, ss))
    s := ss

lastSubResultant(P : polR, Q : polR) := polR ==
zero?(Q) or zero?(P) => 0
if degree(P) < degree(Q) then (P, Q) := (Q, P)
zero?(degree(Q)) => (LC(Q)**degree(P))::polR
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, -Q))
Z : polR := P
repeat
  -- Z = S_d (except the first turn : Z = P)
  -- P = S_{c-1} ~ S_d, Q = S_{d-1}, s = lc(S_d)
  zero?(Q) => return Z
  Z := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
  -- Z = S_{e-1} ~ S_{d-1}
  zero?(degree(Z)) => return Z
(P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

lastSubResultantEuclidean(P : polR, Q : polR) :
  Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
  if degree(P) < degree(Q) then
    l := lastSubResultantEuclidean(Q, P)
    return construct(l.coef2, l.coef1, l.subResultant)
  if zero?(degree(Q)) then
    degP : NNI := degree(P)
    zero?(degP) =>
      error("lastSubResultantEuclidean$PRS : constant polynomials")
    s : R := LC(Q)**(degP-1)::NNI
    return construct(1.coef2, 1.coef1, 1.subResultant)
  if zero?(degree(Q)) then
    degP : NNI := degree(P)
    zero?(degP) =>
      s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 0::polR, 1::polR]
    pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
    VZ : Vector(polR) := copy(VP)
    repeat
      -- VZ.1 = S_d, VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = lc(S_d)
      -- S_{c-1} = VP.2 P_0 + VP.3 Q_0
      -- S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
      -- S_d = VZ.2 P_0 + VZ.3 Q_0
      (Q, Z) := (VQ.1, VZ.1)
      zero?(Q) => return construct(VZ.2, VZ.3, VZ.1)
      VZ := Lazard3(VQ, LC(Q), s, (degree(Z) - degree(Q))::NNI)
      zero?(degree(Z)) => return construct(VZ.2, VZ.3, VZ.1)
      ss : R := LC(VZ.1)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss
  semiLastSubResultantEuclidean(P : polR, Q : polR) :
    Record(coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
    degree(P) < degree(Q) =>
error("semiLastSubResultantEuclidean$PRS : bad degrees")
if zero?(degree(Q)) then
  degP : NNI := degree(P)
  zero?(degP) =>
    error("semiLastSubResultantEuclidean$PRS : constant polynomials")
  s : R := LC(Q)**(degP-1)::NNI
  return construct(s::polR, s * Q)
if zero?(degree(Q)) then
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
VZ : Vector(polR) := copy(VP)
repeat
  -- VZ.1 = S_d,  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s = lc(S_d)
  -- S_{c-1} =  ...  P_0 + VP.2 Q_0
  -- S_{d-1} =  ...  P_0 + VQ.2 Q_0
  -- S_d =  ...  P_0 + VZ.2 Q_0

  (Q, Z) := (VQ.1, VZ.1)
  zero?(Q) => return construct(VZ.2, VZ.1)
  VZ := Lazard3(VQ, LC(Q), s, (degree(Z) - degree(Q))::NNI)
  zero?(degree(Q)) => return construct(VZ.2, VZ.1)
  ss := R := LC(VZ.1)
  (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

chainSubResultants(P : polR, Q : polR) : List(polR) ==
  zero?(Q) or zero?(P) => []
  if degree(P) < degree(Q) then
    (P, Q) := (Q, P)
    if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
    L := List(polR) := []
  zero?(degree(Q)) => L
  L := [Q]
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  (P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
  -- P = S_{c-1},  Q = S_{d-1},  s = lc(S_d)
  -- L = [S_d,....,S_{q-1}]
  zero?(Q) => return L
  L := concat(Q, L)
  -- L = [S_{d-1},....,S_{q-1}]
  L := concat(Q, L)
  delta : NNI := (degree(P) - degree(Q))::NNI
  Z : polR := Lazard2(Q, LC(Q), s, delta)
  if delta > 1 then L := concat(Z, L)
  -- L = [S_e,....,S_{q-1}]
  zero?(degree(Z)) => return L
  (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

schema(P : polR, Q : polR) : List(NNI) ==
zero?(Q) or zero?(P) => []
if degree(P) < degree(Q) then (P, Q) := (Q, P)
zero?(degree(Q)) => [0]
L : List(NNI) := []
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, Q))
repeat
  -- P = S_{c-1}, Q = S_{d-1}, s = lc(S_d)
  zero?(Q) => return L
  e : NNI := degree(Q)
  L := concat(e, L)
  delta : NNI := (degree(P) - e)::NNI
  Z : polR := Lazard2(Q, LC(Q), s, delta) -- Z = S_e \ldots S_{d-1}
  if delta > 1 then L := concat(e, L)
  zero?(e) => return L
  (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

subResultantGcd(P : polR, Q : polR) : polR ==
  zero?(P) and zero?(Q) => 0
  zero?(P) => Q
  zero?(Q) => P
  if degree(P) < degree(Q) then (P, Q) := (Q, P)
  zero?(degree(Q)) => 1$polR
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  (P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
  -- P = S_{c-1}, Q = S_{d-1}, s = lc(S_d)
  zero?(Q) => return P
  zero?(degree(Q)) => return 1$polR
  Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
  -- Z = S_e \ldots S_{d-1}
  (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

subResultantGcdEuclidean(P : polR, Q : polR) : Record(coef1 : polR, coef2 : polR, gcd : polR) ==
  zero?(P) and zero?(Q) => construct(0::polR, 0::polR, 0::polR)
  zero?(P) => construct(0::polR, 1::polR, Q)
  zero?(Q) => construct(1::polR, 0::polR, P)
  if degree(P) < degree(Q) then
    l := subResultantGcdEuclidean(Q, P)
    return construct(l.coef2, l.coef1, l.gcd)
  zero?(degree(Q)) => construct(0::polR, 1::polR, Q)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  VP : Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
  VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
  -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
-- S_{c-1} = VP.2 P_0 + VP.3 Q_0,  S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
(P, Q) := (VP.1, VQ.1)
zero?(Q) => return construct(VP.2, VP.3, P)
e : NNI := degree(Q)
zero?(e) => return construct(VQ.2, VQ.3, Q)
ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

semiSubResultantGcdEuclidean2(P: polR, Q: polR) :
  Record(coef2 : polR, gcd : polR) ==
  zero?(P) and zero?(Q) => construct(0::polR, 0::polR)
  zero?(P) => construct(1::polR, Q)
  zero?(Q) => construct(0::polR, P)
  degree(P) < degree(Q) =>
    error("semiSubResultantGcdEuclidean2$PRS : bad degrees")
zero?(degree(Q)) => construct(1::polR, Q)
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
  VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
  repeat
    -- P=S_{c-1}, Q=S_{d-1}, s=lc(S_d)
    -- S_{c-1} = ? P_0 + old_cf2 Q_0,  S_{d-1} = ? P_0 + cf2 Q_0
    (P, Q) := (VP.1, VQ.1)
zero?(Q) => return construct(VP.2, P)
e : NNI := degree(Q)
zero?(e) => return construct(VQ.2, Q)
  ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
  (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss
semiSubResultantGcdEuclidean1(P: polR, Q: polR) :
  Record(coef1 : polR, gcd : polR) ==
  result := subResultantGcdEuclidean(P,Q)
  [result.coef1, result.gcd]

discriminant(P: polR) : R ==
d := degree(P)
  zero?(d) => error "cannot take discriminant of constants"
a := (d * (d-1)) quo 2
a := (-1)**a::NonNegativeInteger
dP := polR := differentiate P
r : R := resultant(P, dP)
d := d - degree(dP) - 1
return (if zero?(d) then a * (r exquo LC(P))::R
  else a * r * LC(P)**(d-1)::NNI)

discriminantEuclidean(P : polR) :
  Record(coef1 : polR, coef2 : polR, discriminant : R) ==
d : Integer := degree(P)
zero?(d) => error "cannot take discriminant of constants"
a : Integer := (d * (d-1)) quo 2
a := (-1)**a::NonNegativeInteger
dP : polR := differentiate P
rE := resultantEuclidean(P, dP)
d := d - degree(dP) - 1
if zero?(d) then
  c1 : polR := a * (rE.coef1 exquo LC(P))::polR
c2 : polR := a * (rE.coef2 exquo LC(P))::polR
cr : R := a * (rE.resultant exquo LC(P))::R
else
  c1 : polR := a * rE.coef1 * LC(P)**(d-1)::NNI
c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
return construct(c1, c2, cr)

semiDiscriminantEuclidean(P : polR) :
Record(coef2 : polR, discriminant : R) ==
d : Integer := degree(P)
zero?(d) => error "cannot take discriminant of constants"
a : Integer := (d * (d-1)) quo 2
a := (-1)**a::NonNegativeInteger
dP : polR := differentiate P
rE := semiResultantEuclidean2(P, dP)
d := d - degree(dP) - 1
if zero?(d) then
  c2 : polR := a * (rE.coef2 exquo LC(P))::polR
cr : R := a * (rE.resultant exquo LC(P))::R
else
  c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
return construct(c2, cr)

if R has GcdDomain then
resultantReduit(P : polR, Q : polR) : R ==
UV := subResultantGcdEuclidean(P, Q)
UVs : polR := UV.gcd
degree(UVs) > 0 => 0
l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
return (LC(UVs) exquo gcd(l))::R

resultantReduitEuclidean(P : polR, Q : polR) :
Record(coef1 : polR, coef2 : polR, resultantReduit : R) ==
UV := subResultantGcdEuclidean(P, Q)
UVs : polR := UV.gcd
degree(UVs) > 0 => construct(0::polR, 0::polR, 0::R)
l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
gl : R := gcd(l)
c1 : polR := (UV.coef1 exquo gl)::polR
c2 : polR := (UV.coef2 exquo gl)::polR
rr : R := (LC(UVs) exquo gl)::R
return construct(c1, c2, rr)

semiResultantReduitEuclidean(P : polR, Q : polR) :
    Record(coef2 : polR, resultantReduit : R) ==
    UV := subResultantGcdEuclidean(P, Q)
    UVs : polR := UV.gcd
    degree(UVs) > 0 => construct(0::polR, 0::R)
    l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
    gl : R := gcd(l)
    c2 : polR := (UV.coef2 exquo gl)::polR
    rr : R := (LC(UVs) exquo gl)::R
    return construct(c2, rr)

gcd_naif(P : polR, Q : polR) : polR ==
    -- valid over a field
    zero?(P) => (Q exquo LC(Q))::polR
    repeat
        zero?(Q) => return (P exquo LC(P))::polR
        zero?(degree(Q)) => return 1$polR
        (P, Q) := (Q, divide(P, Q).remainder)
    end repeat
    gcd(P : polR, Q : polR) : polR ==
    R has Finite => gcd_naif(P, Q)
    zero?(P) => Q
    zero?(Q) => P
cP : R := content(P)
cQ : R := content(Q)
P := (P exquo cP)::polR
Q := (Q exquo cQ)::polR
G : polR := subResultantGcd(P, Q)
return gcd(cP,cQ) * primitivePart(G)

---

—PRS.dotabb—

"PRS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PRS" -> "PFECAT"
package INTPAF PureAlgebraicIntegration

— PureAlgebraicIntegration.input —

)set break resume
)sys rm -f PureAlgebraicIntegration.output
)spool PureAlgebraicIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PureAlgebraicIntegration
--E 1

)spool
)lisp (bye)

— PureAlgebraicIntegration.help —

====================================================================
PureAlgebraicIntegration examples
====================================================================

Integration of pure algebraic functions.

This package provides functions for integration, limited integration, extended integration and the risch differential equation for pure algebraic integrands.

See Also:
  o )show PureAlgebraicIntegration
PureAlgebraicIntegration (INTPAF)

Exports:
palgLODE palgRDE palgextint palgint palglimint

— package INTPAF PureAlgebraicIntegration —

)abbrev package INTPAF PureAlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 27 May 1988
++ Date Last Updated: 24 June 1994
++ Description:
++ Integration of pure algebraic functions;
++ This package provides functions for integration, limited integration,
++ extended integration and the risch differential equation for
++ pure algebraic integrands;

PureAlgebraicIntegration(R, F, L): Exports == Implementation where
  R: Join(GcdDomain,RetractableTo Integer,OrderedSet,CharacteristicZero,
    LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, AlgebraicallyClosedField,
    TranscendentalFunctionCategory)
  L: SetCategory
SY ==> Symbol
N ==> NonNegativeInteger
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
UPUP==> SparseUnivariatePolynomial RF
IR ==> IntegrationResult F
IR2 ==> IntegrationResultFunctions2(curve, F)
ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
LDALG ==> LinearOrdinaryDifferentialOperator1 curve
RDALG ==> PureAlgebraicLODE(F, UP, UPUP, curve)
LOG ==> Record(coeff:F, logand:F)
CHAPTER 17. \textit{CHAPTER P}

\textbf{Rec} \Rightarrow \text{Record\,(particular:U1, basis:\text{List}\ F)}

\textbf{Cnd} \Rightarrow \text{Record\,(left:UP, right:UP)}

\textbf{Chv} \Rightarrow \text{Record\,(int:\text{UPUP}, left:UP, right:UP, den:RF, deg:N)}

\textbf{U1} \Rightarrow \text{Union\,(F, \{failed\})}

\textbf{U2} \Rightarrow \text{Union\,(Record\,(ratpart:F, coeff:F),\{failed\})}

\textbf{U3} \Rightarrow \text{Union\,(Record\,(mainpart:F, limitedlogs:\text{List}\ LOG), \{failed\})}

\textbf{Fail} \Rightarrow \text{error \{failed - cannot handle that integrand\}}

\textbf{Exports} \Rightarrow \text{with}

\textbf{Palgint} : (F, K, K) \rightarrow \text{IR}

++ \text{palgint\,(f, x, y) returns the integral of \text{\spad{f(x,y)dx} where y is an algebraic function of x.}}

\textbf{Palgextint}: (F, K, K, F) \rightarrow \text{U2}

++ \text{palgextint\,(f, x, y, g) returns functions \text{\[h, c\]} such that}
++ \text{\spad{dh/dx = f(x,y) - c g}, where y is an algebraic function of x;}
++ \text{return \{failed\} if no such functions exist.}

\textbf{Palglimint}: (F, K, K, List\ F) \rightarrow \text{U3}

++ \text{palglimint\,(f, x, y, [u1,\ldots,un]) returns functions}
++ \text{\spad{\{h, [[ci, ui]]\}} such that the ui\,'s are among \text{\spad{\{u1,\ldots,un\}}}
++ \text{and \spad{d(h + sum(ci log(ui)))/dx = f(x,y)} if such functions exist,}
++ \text{\{failed\} otherwise;}
++ \text{y is an algebraic function of x.}

\textbf{PalgRDE} : (F, F, F, K, K, (F, F, SY) \rightarrow \text{U1}) \rightarrow \text{U1}

++ \text{palgRDE\,(nfp, f, g, x, y, foo) returns a function \text{\spad{z(x,y)}}}
++ \text{such that \spad{dz/dx + n * df/dx z(x,y) = g(x,y)} if such a z exists,}
++ \text{\{failed\} otherwise;}
++ \text{y is an algebraic function of x;}
++ \text{\spad{foo(a, b, x)} is a function that solves}
++ \text{\spad{du/dx + n * da/dx u(x) = u(x)}}
++ \text{for an unknown \spad{u(x)} not involving y.}
++ \text{\spad{nfip} is \spad{n * df/dx}.}

\text{if L has LinearOrdinaryDifferentialOperatorCategory F then}

\textbf{PalgLODE} : (L, F, K, K, SY) \rightarrow \text{REC}

++ \text{palgLODE\,(op, g, kx, y, x) returns the solution of \text{\spad{op f = g}.}}
++ \text{y is an algebraic function of x.}

\textbf{Implementation} \Rightarrow \text{add}

\text{import IntegrationTools\,(R, F)}
\text{import RationalIntegration\,(F, UP)}
\text{import GenusZeroIntegration\,(R, F, L)}
\text{import ChangeOfVariable\,(F, UP, UPUP)}
\text{import IntegrationResultFunctions2\,(F, F)}
\text{import IntegrationResultFunctions2\,(RF, F)}
\text{import SparseUnivariatePolynomialFunctions2\,(F, RF)}
\text{import UnivariatePolynomialCommonDenominator\,(UP, RF, UPUP)}
\text{import PolynomialCategoryQuotientFunctions\,(\text{IndexedExponents K, K, R, P, F})}

\textbf{QuadIffCan} : (K, K) \rightarrow \text{Union\,(Record\,(coef:F, poly:UP), \{failed\})}
\textbf{LinearInXIfCan} : (K, K) \rightarrow \text{Union\,(Record\,(xsub:F, dxsub:RF), \{failed\})}
prootintegrate : (F, K, K) -> IR
prootintegrate1: (UPUP, K, K, UPUP) -> IR
prootextint : (F, K, K, F) -> U2
prootlimint : (F, K, K, List F) -> U3
prootRDE : (F, F, F, K, K, (F, F, SY) -> U1) -> U1
palgRDE1 : (F, F, K, K) -> U1
palgLODE1 : (List F, F, K, K, SY) -> REC
palgintegrate : (F, K, K) -> IR
palgext : (F, K, K, F) -> U2
palglim : (F, K, K, List F) -> U3
UPUP2F1 : (UPUP, RF, RF, K, K) -> F
UPUP2F0 : (UPUP, K, K) -> F
RF2UPUP : (RF, UPUP) -> UPUP
algaddx : (IR, F) -> IR
chvarIfCan : (UPUP, RF, UP, RF) -> Union(UPUP, "failed")
changeVarIfCan : (UPUP, RF, N) -> Union(CHV, "failed")
rationalInt : (UPUP, N, UP) -> IntegrationResult RF
chv : (UPUP, N, F, F) -> RF
chv0 : (UPUP, N, F, F) -> F
candidates : UP -> List CND

dummy := new()$SY
dumk := kernel(dummy)$K

UPUP2F1(p, t, cf, kx, k) == UPUP2F0(eval(p, t, cf), kx, k)
UPUP2F0(p, kx, k) == multivariate(p, kx, k::F)
chv(f, n, a, b) == univariate(chv0(f, n, a, b), dumk)

RF2UPUP(f, modulus) ==
  bc := extendedEuclidean(map((z1:F):RF+->z1::UP::RF, denom f), modulus, 1)::Record(coef1:UPUP, coef2:UPUP)
  (map((x1:F):RF+->x1::UP::RF, numer f) * bc.coef1) rem modulus

-- returns "failed", or (xx, c) such that f(x, y)dx = f(xx, y) c dy
-- if p(x, y) = 0 is linear in x
linearInXIfCan(x, y) ==
  a := b := 0$UP
  p := clearDenominator lift(minPoly y, x)
  while p ^= 0 repeat
    degree(q := numer leadingCoefficient p) > 1 => return "failed"
    a := a + monomial(coefficient(q, 1), d := degree p)
    b := b - monomial(coefficient(q, 0), d)
    p := reductum p
  xx:RF := b / a
  [xx(dumk::F), differentiate(xx, differentiate)]

-- return Int(f(x,y)dx) where y is an n"th root of a rational function in x
prootintegrate(f, x, y) ==
  modulus := lift(p := minPoly y, x)
  rf := reductum(ff := univariate(f, x, y, p))
\[(r := \text{retractIfCan}(\text{rf})@\text{Union}(\text{RF}, \text{"failed"})) \quad \text{case RF} \quad \text{and rf} \neq 0 \Rightarrow\]

\[\quad \text{-- in this case, ff := lc(ff) y^i + r so we we integrate both terms}\]

\[\quad \text{-- separately to gain time}\]

\[\quad \text{map}(f1+->f1(x::F), \text{integrate}(r::\text{RF}))+\]

\[\quad \text{prootintegrate1(leadingMonomial ff, x, y, modulus)}\]

\[\text{prootintegrate1}(ff, x, y, \text{modulus}) =\]

\[\text{chv:CHV}\]

\[\quad r := \text{radPoly} (\text{modulus})::\text{Record}(\text{radicand:RF, deg:N})\]

\[\quad (uu := \text{changeVarIfCan}(ff, r.\text{radicand}, r.\text{deg})) \quad \text{case CHV} \Rightarrow\]

\[\quad \text{chv := uu::CHV}\]

\[\quad \text{newalg := nthRoot}(\text{((chv.left)}(\text{dumk::F}), \text{chv.deg})\]

\[\quad \text{kvz := retract(numer newalg)@K}\]

\[\quad \text{newf := multivariate(chv.\text{int}, ku := dumk, newalg)}\]

\[\quad \text{vu := ((chv.right)}(x::F)\]

\[\quad \text{vz := (chv.den)(x::F) * (y::F) * denom(newalg)::F}\]

\[\quad \text{map}(x1+->\text{eval}(x1, [ku, kz], [vu, vz]), \text{palgint(newf, ku, kz)})\]

\[\text{cv := chvar(ff, modulus)}\]

\[\quad r := \text{radPoly(cv.poly)}::\text{Record}(\text{radicand:RF, deg:N})\]

\[\quad \text{qprime := differentiate}(q := \text{retract}(r.\text{radicand})@\text{UP})::\text{RF}\]

\[\quad \text{not zero? qprime and}\]

\[\quad ((u := \text{chvarIfCan}(cv.\text{func}, 1, \text{q, inv qprime})) \quad \text{case UPUP} \Rightarrow\]

\[\quad \quad \text{m := monomial}(1, r.\text{deg})@\text{UPUP} - q::\text{RF}@\text{UPUP}\]

\[\quad \quad \text{map}(x1+->\text{UPUP2F1}(\text{RF2UPUP}(x1, m), cv.\text{c1}, cv.\text{c2}, x, y),\]

\[\quad \quad \quad \text{rationalInt}(u::\text{UPUP}, r.\text{deg}, \text{monomial}(1, 1))\]

\[\quad \text{curve} := \text{RadicalFunctionField}(F, \text{UP}, \text{UPUP}, q::\text{RF}, r.\text{deg})\]

\[\quad \text{algaddx}(\text{map}(x1+->\text{UPUP2F1}(\text{lift x1, cv.\text{c1}, cv.\text{c2}, x, y}),\]

\[\quad \quad \quad \text{palgintegrate}(\text{reduce}(cv.\text{func}), \text{differentiate@UP}@\text{ALG}@\text{IR2}, x::F)\]

-- Do the rationalizing change of variable

-- \(\int(f(x, y) \, dx) \to \int(n \, u^{(n-1)} f((u^n - b)/a, u) / a \, du)\) where

-- \(u^n = y^n = g(x) = a x + b\)

-- returns the integral as an integral of a rational function in \(u\)

\[\text{rationalInt}(f, n, g) =\]

-- not one? degree g => error "rationalInt: radicand must be linear"

not ((degree g) = 1) \Rightarrow error "rationalInt: radicand must be linear"

\[\quad a := \text{leadingCoefficient g}\]

\[\quad \text{integrate}(n * \text{monomial}(\text{inv a}, (n-1)::N)@\text{UP}\]

\[\quad \quad \quad \text{* chv(f, n, a, \text{leadingCoefficient reductum g})}\]

-- Do the rationalizing change of variable \(f(x,y) \to f((u^n - b)/a, u)\) where

-- \(u = y = (a x + b)^{1/n}\).

-- Returns \(f((u^n - b)/a,u)\) as an element of \(F\)

\[\text{chv0}(f, n, a, b) =\]

\[\quad d := \text{dumk::F}\]

\[\quad (f \cdot (d::\text{UP}@\text{RF})) ((d ** n - b) \text{ / a})\]

-- candidates(p) returns a list of pairs \([g, u]\) such that \(p(x) = g(u(x))\),

-- those \(u\)'s are candidates for change of variables
-- currently uses a dumb heuristic where the candidates u's are p itself
-- and all the powers x^2, x^3, ..., x^deg(p),
-- will use polynomial decomposition in smarter days MB 8/93
candidates p ==
  l:List(CND) := empty()
  ground? p => l
  for i in 2..degree p repeat
    if (u := composite(p, xi := monomial(1, i))) case UP then
      l := concat([u::UP, xi], l)
    concat([monomial(1, 1), p], l)
-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x,
-- z = d y for some polynomial d, and z^m = g(u)
-- returns either [r(u, z), g, u, d, m] or "failed"
-- we have y^n = radi
changeVarIfCan(p, radi, n) ==
  rec := rootPoly(radi, n)
  for cnd in candidates(rec.radicand) repeat
    (u := chvarIfCan(p, rec.coef, cnd.right,
      inv(differentiate(cnd.right)::RF))) case UPUP =>
      return [u::UPUP, cnd.left, cnd.right, rec.coef, rec.exponent]
"failed"
-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x and z = d y
-- we have y^n = a(x)/d(x)
-- returns either "failed" or r(u, z)
chvarIfCan(p, d, u, u1) ==
  ans:UPUP := 0
  while p ^= 0 repeat
    (v := composite(u1 * leadingCoefficient(p) / d ** degree(p), u))
    case "failed" => return "failed"
    ans := ans + monomial(v::RF, degree p)
    p := reductum p
  ans
algaddx(i, xx) ==
  elem? i => i
  mkAnswer(ratpart i, logpart i,
    [[- ne.integrand / (xx**2), xx] for ne in notelem i])
prootRDE(nfp, f, g, x, k, rde) ==
  modulus := lift(p := minPoly k, x)
  r := radPoly(modulus)::Record(radicand:RF, deg:N)
  rec := rootPoly(r.radicand, r.deg)
  dqdx := inv(differentiate(q := rec.radicand)::RF)
  (uf := chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,1)) case UPUP and
  (ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP =>
  (u := rde(chv0(uf::UPUP, rec.exponent, 1, 0), rec.exponent *
(dumk::F) ** (rec.exponent * (rec.exponent - 1))
* chv0(ug::UPUP, rec.exponent, 1, 0),
  symbolIfCan(dumk::SY)) case "failed" => "failed"

eval(u::F, dumk, k::F)
-- one?(rec.coef) =>
((rec.coef) = 1) =>
  curve := RadicalFunctionField(F, UP, UPUP, q::RF, rec.exponent)
  rc := algDsolve(D()$LDALG + reduce(univariate(nfp, x, k, p))::LDALG,
                  reduce univariate(g, x, k, p))$RDALG
  rc.particular case "failed" => "failed"
  UPUP2F0(lift((rc.particular)::curve), x, k)
palgRDE1(nfp, g, x, k)

prootlimint(f, x, k, lu) ==
  modulus := lift(p := minPoly k, x)
  r := radPoly(modulus)::Record(radicaland:RF, deg:N)
  rec := rootPoly(r.radicaland, r.deg)
  dqdx := inv(differentiate(q := rec.radicaland)::RF)
  (uf:=chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP =>
    l := empty()$List(RF)
    n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
    for u in lu repeat
      if ((v:=chvarIfCan(uu:=univariate(u,x,k,p),rec.coef,q,dqdx)) case UPUP)
        then l := concat(n * chv(v::UPUP,rec.exponent,1,0), l) else FAIL
    m := monomial(1, rec.exponent)$UP - q::RF::UPUP
    map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
    limitedint(n * chv(uf::UPUP, rec.exponent, 1, 0), reverse_! l))
  cv := chvar(ff, modulus)
  r := radPoly(cv.poly)::Record(radicaland:RF, deg:N)
  dqdx := inv(differentiate(q := retract(r.radicaland)::UP)::RF)
  curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
  (ui := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
  case "failed" => FAIL
  [UPUP2F1(lift(ui::curve), cv.c1, cv.c2, x, k), empty()]

prootextint(f, x, k, g) ==
  modulus := lift(p := minPoly k, x)
  r := radPoly(modulus)::Record(radicaland:RF, deg:N)
  rec := rootPoly(r.radicaland, r.deg)
  dqdx := inv(differentiate(q := rec.radicaland)::RF)
  (ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP and
  (uf:=chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP =>
    m := monomial(1, rec.exponent)$UP - q::RF::UPUP
    n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
    map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
    extendedint(n * chv(uf::UPUP, rec.exponent, 1, 0),
    n * chv(ug::UPUP, rec.exponent, 1, 0)))
  cv := chvar(ff, modulus)
  r := radPoly(cv.poly)::Record(radicaland:RF, deg:N)
  dqdx := inv(differentiate(q := retract(r.radicaland)::UP)::RF)
curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
(u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
case "failed" => FAIL
[UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

case "failed" => FAIL
[UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

palgRDE1(nfp, g, x, y) ==
palgLODE1([nfp, 1], g, x, y, symbolIfCan(x)::SY).particular

palgLODE1(eq, g, kx, y) ==
modulus:= lift(p := minPoly y, kx)
curve := AlgebraicFunctionField(F, UP, UPUP, modulus)
neq:LDALG := 0
for f in eq for i in 0.. repeat
  neq := neq + monomial(reduce univariate(f, kx, y, p), i)
empty? remove_!(y, remove_!(kx, varselect(kernels g, x))) =>
rec := algDsolve(neq, reduce univariate(g, kx, y, p))$RDALG
bas:List(F) := [UPUP2F0(lift h, kx, y) for h in rec.basis]
rec.particular case "failed" => ["failed", bas]
[UPUP2F0(lift((rec.particular)::curve), kx, y), bas]
rec := algDsolve(neq, 0)
["failed", [UPUP2F0(lift h, kx, y) for h in rec.basis]]

palgintegrate(f, x, k) ==
modulus:= lift(p := minPoly y, k)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
alggaddx(map(x1+->UPUP2F1(lift x1, cv.c1, cv.c2, x, k),
palgintegrate(reduce(cv.func), differentiate$UP)$ALG)$IR2, x::F)

palglim(f, x, k, lu) ==
modulus:= lift(p := minPoly y, k)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
(u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
case "failed" => FAIL
[UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), empty()]

palgext(f, x, k, g) ==
modulus:= lift(p := minPoly y, k)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
(u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
case "failed" => FAIL
[UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

palgint(f, x, y) ==
(v := linearInXIfCan(x, y)) case "failed" =>
(u := quadIfCan(x, y)) case "failed" =>
  is?(y, "nthRoot"::SY) => prootintegrate(f, x, y)
  is?(y, "rootOf"::SY) => palgintegrate(f, x, y)
FAIL
palgint0(f, x, y, u.coef, u.poly)
palgint0(f, x, y, dumk, v.xsub, v.dxsub)

palgextint(f, x, y, g) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
  is?(y, "nthRoot"::SY) => prootextint(f, x, y, g)
  is?(y, "rootOf"::SY) => palgext(f, x, y, g)
FAIL
palgextint0(f, x, y, g, u.coef, u.poly)
palgextint0(f, x, y, g, dumk, v.xsub, v.dxsub)

palglimint(f, x, y, lu) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
  is?(y, "nthRoot"::SY) => prootlimint(f, x, y, lu)
  is?(y, "rootOf"::SY) => palglim(f, x, y, lu)
FAIL
palglimint0(f, x, y, lu, u.coef, u.poly)
palglimint0(f, x, y, lu, dumk, v.xsub, v.dxsub)

palgRDE(nfp, f, g, x, y, rde) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
  is?(y, "nthRoot"::SY) => prootRDE(nfp, f, g, x, y, rde)
  palgRDE1(nfp, g, x, y)
  palgRDE0(f, g, x, y, rde, u.coef, u.poly)
palgRDE0(f, g, x, y, rde, dumk, v.xsub, v.dxsub)

-- returns "failed", or (d, P) such that (dy)**2 = P(x)
-- and degree(P) = 2
quadIfCan(x, y) ==
  (degree(p := minPoly y) = 2) and zero?(coefficient(p, 1)) =>
  d := denom(ff :=
    univariate(- coefficient(p, 0) / coefficient(p, 2), x))
  degree(radi := d * numer ff) = 2 => [d(x::F), radi]
  "failed"
"failed"

if L has LinearOrdinaryDifferentialOperatorCategory F then
  palgLODE(eq, g, kx, y, x) ==
    (v := linearInXIfCan(kx, y)) case "failed" =>
    (u := quadIfCan(kx, y)) case "failed" =>
    palgLODE1([coefficient(eq, i) for i in 0..degree eq], g, kx, y, x)
    palgLODE0(eq, g, kx, y, u.coef, u.poly)
  palgLODE0(eq, g, kx, y, dumk, v.xsub, v.dxsub)
package ODEPAL PureAlgebraicLODE

-- PureAlgebraicLODE.input --

)set break resume
)sys rm -f PureAlgebraicLODE.output
)spool PureAlgebraicLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PureAlgebraicLODE
--E 1

)spool
)lisp (bye)

-- PureAlgebraicLODE.help --

====================================================================

In-field solution of a linear ordinary differential equation, pure algebraic case.

See Also:
PureAlgebraicLODE (ODEPAL)

Exports:
algDsolve

— package ODEPAL PureAlgebraicLODE —

)abbrev package ODEPAL PureAlgebraicLODE
++ Author: Manuel Bronstein
++ Date Created: 21 August 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ In-field solution of an linear ordinary differential equation,
++ pure algebraic case.

PureAlgebraicLODE(F, UP, UPUP, R):Exports == Implementation where
  F : Join(Field, CharacteristicZero,
              RetractableTo Integer, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R : FunctionFieldCategory(F, UP, UPUP)

RF ==> Fraction UP
V ==> Vector RF
U ==> Union(R, "failed")
REC ==> Record(particular: Union(RF, "failed"), basis: List RF)
L ==> LinearOrdinaryDifferentialOperator1 R
LQ ==> LinearOrdinaryDifferentialOperator1 RF
Exports ==> with
  algDsolve: (L, R) -> Record(particular: U, basis: List R)
  ++ algDsolve(op, g) returns \spad{["failed", []]} if the equation
  ++ \spad{op y = g} has no solution in \spad{R}. Otherwise, it returns
  ++ \spad{[f, [y_1,...,y_m]]} where \spad{f} is a particular rational
  ++ solution and the \spad{y_i's} form a basis for the solutions in
  ++ \spad{R} of the homogeneous equation.

Implementation ==> add
  import RationalLODE(F, UP)
  import SystemODESolver(RF, LQ)
  import ReduceLODE(RF, LQ, UPUP, R, L)

algDsolve(l, g) ==
  rec := reduceLODE(l, g)
  sol := solveInField(rec.mat, rec.vec, ratDsolve)
  bas:List(R) := [represents v for v in sol.basis]
  (u := sol.particular) case V => [represents(u::V), bas]
  ["failed", bas]
PushVariables examples
====================================================================
This package has no description
See Also:
  o )show PushVariables

PushVariables (PUSHVAR)

Exports:

package PUSHVAR PushVariables

PushVariables(R,E,OV,PPR):C == T where
  E : OrderedAbelianMonoidSup
OV: OrderedSet with
  convert: % -> Symbol
  ++ convert(x) converts x to a symbol
variable: Symbol -> Union(%, "failed")
  ++ variable(s) makes an element from symbol s or fails

R : Ring
PR ==> Polynomial R
PPR: PolynomialCategory(PR,E,OV)
SUP ==> SparseUnivariatePolynomial

C == with
  pushdown : (PPR, OV) -> PPR
  ++ pushdown(p,v) \undocumented{}
  pushdown : (PPR, List OV) -> PPR
  ++ pushdown(p,lv) \undocumented{}
  pushup : (PPR, OV) -> PPR
  ++ pushup(p,v) \undocumented{}
  pushup : (PPR, List OV) -> PPR
  ++ pushup(p,lv) \undocumented{}
  map : ((PR -> PPR), PPR) -> PPR
  ++ map(f,p) \undocumented{}

T == add
pushdown(g:PPR,x:OV) : PPR ==
  eval(g,x,monomial(1,convert x,1)$PR)
pushdown(g:PPR, lv:List OV) : PPR ==
  vals:=[monomial(1,convert x,1)$PR for x in lv]
  eval(g,lv,vals)
map(f:(PR -> PPR), p: PPR) : PPR ==
  ground? p => f(retract p)
  v:=mainVariable(p)::OV
  multivariate(map((x:PPR):PPR+->map(f,x),univariate(p,v)),v)

---- push back the variable ----
pushupCoef(c:PR, lv:List OV): PPR ==
  ground? c => c::PPR
  v:=mainVariable(c)::Symbol
  v2 := variable(v)$OV
  uc := univariate(c,v)
  ppr : PPR := 0
  v2 case OV =>
    while not zero? uc repeat
      ppr := ppr + monomial(1,v2,degree(uc))$PPR *
        pushupCoef(leadingCoefficient uc, lv)
      uc := reductum uc
    ppr
  while not zero? uc repeat
    ppr := ppr + monomial(1,v,degree(uc))$PR *
      pushupCoef(leadingCoefficient uc, lv)
uc := reductum uc
   ppr

pushup(f:PPR,x:OV) : PPR ==
   map(y +-> pushupCoef(y, [x]), f)

pushup(g:PPR, lv:List OV) : PPR ==
   map(y +-> pushupCoef(y, lv), g)

— PUSHVAR.dotabb —

"PUSHVAR" [color="#FF4488", href="bookvol10.4.pdf#nameddest=PUSHVAR"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"PUSHVAR" -> "PFECAT"
Chapter 18

Chapter Q

package QALGSET2 QuasiAlgebraicSet2

— QuasiAlgebraicSet2.input —

)set break resume
)sys rm -f QuasiAlgebraicSet2.output
)spool QuasiAlgebraicSet2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuasiAlgebraicSet2
--E 1

)spool
)lisp (bye)

— QuasiAlgebraicSet2.help —

====================================================================
QuasiAlgebraicSet2 examples
====================================================================

QuasiAlgebraicSet2 adds a function radicalSimplify which uses
IdealDecompositionPackage to simplify the representation of a
quasi-algebraic set.
A quasi-algebraic set is the intersection of a Zariski closed set, defined as the common zeros of a given list of polynomials (the defining polynomials for equations), and a principal Zariski open set, defined as the complement of the common zeros of a polynomial f (the defining polynomial for the inequation).

Quasi-algebraic sets are implemented in the domain QuasiAlgebraicSet, where two simplification routines are provided: idealSimplify and simplify.

The function radicalSimplify is added for comparison study only.

Because the domain IdealDecompositionPackage provides facilities for computing with radical ideals, it is necessary to restrict the ground ring to the domain Fraction Integer, and the polynomial ring to be of type DistributedMultivariatePolynomial.

The routine radicalSimplify uses these to compute groebner basis of radical ideals and is inefficient and restricted when compared to the two in QuasiAlgebraicSet.

See Also:
o )show QuasiAlgebraicSet2

---

**QuasiAlgebraicSet2 (QALGSET2)**

![QALGSET2 ALIST](image)

Exports:
radicalSimplify

---

package QALGSET2 QuasiAlgebraicSet2 ---
+++ Description:
+++ \spadtype{QuasiAlgebraicSet2} adds a function \spadfun{radicalSimplify} which uses \spadtype{IdealDecompositionPackage} to simplify the representation of a quasi-algebraic set. A quasi-algebraic set is the intersection of a Zariski closed set, defined as the common zeros of a given list of polynomials (the defining polynomials for equations), and a principal Zariski open set, defined as the complement of the common zeros of a polynomial \( f \) (the defining polynomial for the inequation).
+++ Quasi-algebraic sets are implemented in the domain \spadtype{QuasiAlgebraicSet}, where two simplification routines are provided: \spadfun{idealSimplify} and \spadfun{simplify}.
+++ The function \spadfun{radicalSimplify} is added for comparison study only. Because the domain \spadtype{IdealDecompositionPackage} provides facilities for computing with radical ideals, it is necessary to restrict the ground ring to the domain \spadtype{Fraction Integer}, and the polynomial ring to be of type \spadtype{DistributedMultivariatePolynomial}.
+++ The routine \spadfun{radicalSimplify} uses these to compute groebner basis of radical ideals and is inefficient and restricted when compared to the two in \spadtype{QuasiAlgebraicSet}.

QuasiAlgebraicSet2(vl,nv) : C == T where
vl : List Symbol
nv : NonNegativeInteger
R ==> Integer
F ==> Fraction R
Var ==> OrderedVariableList vl
NNI ==> NonNegativeInteger
Expon ==> DirectProduct(nv,NNI)
Dpoly ==> DistributedMultivariatePolynomial(vl,F)
QALG ==> QuasiAlgebraicSet(F, Var, Expon, Dpoly)
newExpon ==> DirectProduct(#newvl, NNI)
newPoly ==> DistributedMultivariatePolynomial(newvl,F)
newVar ==> OrderedVariableList newvl
Status ==> Union(Boolean,"failed") -- empty or not, or don't know

C == with
   radicalSimplify:QALG -> QALG
   ++ radicalSimplify(s) returns a different and presumably simpler
++ representation of s with the defining polynomials for the
++ equations
++ forming a groebner basis, and the defining polynomial for the
++ inequation reduced with respect to the basis, using
++ using groebner basis of radical ideals
T == add

---- Local Functions ----
ts:=new()$Symbol
newvl:=concat(ts, vl)
tv:newVar:=(variable ts)::newVar
npoly : Dpoly -> newPoly
oldpoly : newPoly -> Union(Dpoly,"failed")
f : Var -> newPoly
g : newVar -> Dpoly

import PolynomialIdeals(F,newExpon,newVar,newPoly)
import GroebnerPackage(F,Expon,Var,Dpoly)
import GroebnerPackage(F,newExpon,newVar,newPoly)
import IdealDecompositionPackage(newvl,#newvl)
import QuasiAlgebraicSet(F, Var, Expon, Dpoly)
import PolynomialCategoryLifting(Expon,Var,Expon,Dpoly)
import PolynomialCategoryLifting(newExpon,newVar,F,newPoly,Dpoly)
f(v:Var):newPoly ==
  variable((convert v)@Symbol)@Union(newVar,"failed")::newVar
  ::newPoly
g(v:newVar):Dpoly ==
  v = tv => 0
  variable((convert v)@Symbol)@Union(Var,"failed")::Var::Dpoly

npoly(p:Dpoly) : newPoly ==
  map(z1 +-> f z1, z2 +-> z2::newPoly, p)

oldpoly(q:newPoly) : Union(Dpoly,"failed") ==
  (x:=mainVariable q) case "failed" => (leadingCoefficient q)::Dpoly
  (x::newVar = tv) => "failed"
  map(z1 +-> g z1, z2 +-> z2::Dpoly, q)

radicalSimplify x ==
  status(x)$QALG = true => x -- x is empty
  z0:=definingEquations x
  n0:=definingInequation x
  t:newPoly:= coerce(tv)$newPoly
  tp:newPoly:= t * (npoly n0) - 1$newPoly
  gen:List newPoly:= concat(tp, [npoly g for g in z0])
  id:=ideal gen
  ngb:=generators radical(id)
  member? (1$newPoly, ngb) => empty()$QALG
  gb:List Dpoly:=nil
  while not empty? ngb repeat
    if ((k:=oldpoly ngb.first) case Dpoly) then gb:=concat(k, gb)
    ngb:=ngb.rest
y := quasiAlgebraicSet(gb, primitivePart normalForm(n0, gb))
setStatus(y, false::Status)

---

-- QALGSET2.dotabb --

"QALGSET2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QALGSET2"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"QALGSET2" -> "ALIST"

---

package QCMPACK QuasiComponentPackage

--- QuasiComponentPackage.input ---

)set break resume
)sys rm -f QuasiComponentPackage.output
)spool QuasiComponentPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuasiComponentPackage
--E 1

)spool
)lisp (bye)

---

-- QuasiComponentPackage.help ---

====================================================================
QuasiComponentPackage examples
====================================================================

A package for removing redundant quasi-components and redundant branches when decomposing a variety by means of quasi-components of regular triangular sets.
See Also:
- )show QuasiComponentPackage

QuasiComponentPackage (QCMPACK)

Exports:
- algebraicSort
- branchIfCan
- infRittWu?
- internalInfRittWu?
- internalSubPolSet?
- internalSubQuasiComponent?
- moreAlgebraic?
- prepareDecompose
- removeSuperfluousCases
- removeSuperfluousQuasiComponents
- startTable!
- subCase?
- subPolSet?
- subQuasiComponent?
- subTriSet?
- supDimElseRittWu?

--- package QCMPACK QuasiComponentPackage ---

)abbrev package QCMPACK QuasiComponentPackage
++ Author: Marc Moreno Maza <marc@nag.co.uk>
++ Date Created: 08/30/1998
++ Date Last Updated: 12/16/1998
++ References:
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ d'extensions simples et resolution des systemes d'equations
++ Description:
A package for removing redundant quasi-components and redundant branches when decomposing a variety by means of quasi-components of regular triangular sets.

**QuasiComponentPackage**\((R,E,V,P,TS)\): Exports == Implementation where

\[ R: \text{GcdDomain} \]
\[ E: \text{OrderedAbelianMonoidSup} \]
\[ V: \text{OrderedSet} \]
\[ P: \text{RecursivePolynomialCategory}(R,E,V) \]
\[ TS: \text{RegularTriangularSetCategory}(R,E,V,P) \]
\[ N: \Rightarrow \text{NonNegativeInteger} \]
\[ Z: \Rightarrow \text{Integer} \]
\[ B: \Rightarrow \text{Boolean} \]
\[ S: \Rightarrow \text{String} \]
\[ LP: \Rightarrow \text{List}\ P \]
\[ PtoP: \Rightarrow P \rightarrow P \]
\[ PS: \Rightarrow \text{GeneralPolynomialSet}(R,E,V,P) \]
\[ PWT: \Rightarrow \text{Record}(\text{val}:P,\text{tower}:TS) \]
\[ BWT: \Rightarrow \text{Record}(\text{val}:\text{Boolean},\text{tower}:TS) \]
\[ LpWT: \Rightarrow \text{Record}(\text{val}:(\text{List}\ P),\text{tower}:TS) \]
\[ Branch: \Rightarrow \text{Record}(\text{eq}\!:\text{List}\ P,\text{tower}:TS,\text{ineq}\!:\text{List}\ P) \]
\[ UBF: \Rightarrow \text{Union}(\text{Branch},\text{"failed"}) \]
\[ Split: \Rightarrow \text{List}\ TS \]
\[ Key: \Rightarrow \text{Record}(\text{left}:TS,\text{right}:TS) \]
\[ Entry: \Rightarrow \text{Boolean} \]
\[ H: \Rightarrow \text{TabulatedComputationPackage}(\text{Key},\text{Entry}) \]
\[ polsetpack: \Rightarrow \text{PolynomialSetUtilitiesPackage}(R,E,V,P) \]

**Exports** == with

\[ \text{startTable}!: (S,S,S) \rightarrow \text{Void} \]
\[ \text{stopTable}!: () \rightarrow \text{Void} \]
\[ \text{supDimElseRittWu}?: (TS,TS) \rightarrow \text{Boolean} \]
\[ \text{algebraicSort}: \text{Split} \rightarrow \text{Split} \]
\[ \text{moreAlgebraic}?: (TS,TS) \rightarrow \text{Boolean} \]
\[ \text{subTriSet}?: (TS,TS) \rightarrow \text{Boolean} \]
++ a sub-set of \texttt{axiom\{us\}}.

\texttt{subPolSet?}: (LP, LP) -> Boolean
++ \texttt{axiom\{subPolSet?\}(lp1,lp2)} returns true iff \texttt{axiom\{lp1\}} is
++ a sub-set of \texttt{axiom\{lp2\}}.

\texttt{internalSubPolSet?}: (LP, LP) -> Boolean
++ \texttt{axiom\{internalSubPolSet?\}(lp1,lp2)} returns true iff \texttt{axiom\{lp1\}} is
++ a sub-set of \texttt{axiom\{lp2\}} assuming that these lists are sorted
++ increasingly w.r.t.
++ \texttt{infRittWu?} from \texttt{RecursivePolynomialCategory}.

\texttt{internalInfRittWu?}: (LP, LP) -> Boolean
++ \texttt{axiom\{internalInfRittWu?\}(lp1,lp2)}
++ is an internal subroutine, exported only for development.

\texttt{infRittWu?}: (LP, LP) -> Boolean
++ \texttt{axiom\{infRittWu?\}(lp1,lp2)}
++ is an internal subroutine, exported only for development.

\texttt{internalSubQuasiComponent?}: (TS,TS) -> Union(Boolean,"failed")
++ \texttt{axiom\{internalSubQuasiComponent?\}(ts,us)} returns a
++ boolean \texttt{b} value if the fact that the regular
++ zero set of \texttt{axiom\{us\}} contains that of
++ \texttt{axiom\{ts\}} can be decided (and in that case \texttt{axiom\{b\}} gives this
++ inclusion) otherwise returns \texttt{axiom\{"failed"\}}.

\texttt{subQuasiComponent?}: (TS,TS) -> Boolean
++ \texttt{axiom\{subQuasiComponent?\}(ts,us)} returns true iff
++ \texttt{internalSubQuasiComponent?}
++ returns true.

\texttt{subQuasiComponent?}: (TS,Split) -> Boolean
++ \texttt{axiom\{subQuasiComponent?\}(ts,lus)} returns true iff
++ \texttt{subQuasiComponent?}(ts,us) holds for one \texttt{spad\{us\}
++ in \texttt{spad\{lus\}}.

\texttt{removeSuperfluousQuasiComponents}: Split -> Split
++ \texttt{axiom\{removeSuperfluousQuasiComponents\}(lts)} removes
++ from \texttt{axiom\{lts\}} any \texttt{spad\{ts\}} such that
++ \texttt{subQuasiComponent?\}(ts,us) holds for
++ another \texttt{spad\{us\}} in \texttt{axiom\{lts\}}.

\texttt{subCase?}: (LpWT,LpWT) -> Boolean
++ \texttt{axiom\{subCase?\}(lpwt1,lpwt2)}
++ is an internal subroutine, exported only for development.

\texttt{removeSuperfluousCases}: List LpWT -> List LpWT
++ \texttt{axiom\{removeSuperfluousCases\}(llpwt)}
++ is an internal subroutine, exported only for development.

\texttt{prepareDecompose}: (LP, List(TS),B,B) -> List Branch
++ \texttt{axiom\{prepareDecompose\}(lp,lts,b1,b2)}
++ is an internal subroutine, exported only for development.

\texttt{branchIfCan}: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
++ \texttt{axiom\{branchIfCan\}(leq,ts,lineq,bl,b2,b3,b4,b5)}
++ is an internal subroutine, exported only for development.

\texttt{Implementation == add}

\texttt{squareFreeFactors(lp: LP): LP ==}
lsflp: LP := []
for p in lp repeat
   lsfp := squareFreeFactors(p)$polsetpack
   lsflp := concat(lsfp,lsflp)
sort(infRittWu?,removeDuplicates lsflp)

startTable!(ok: S, ko: S, domainName: S): Void ==
   initTable!()$H
   if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
   if (not empty? domainName) then startStats!(domainName)$H
   void()

stopTable!(): Void ==
   if makingStats?()$H then printStats!()$H
   clearTable!()$H

supDimElseRittWu? (ts:TS,us:TS): Boolean ==
   #ts < #us => true
   #ts > #us => false
   lp1 :LP := members(ts)
   lp2 :LP := members(us)
   while (not empty? lp1) and (not infRittWu?(first(lp2),first(lp1))) repeat
      lp1 := rest lp1
      lp2 := rest lp2
   not empty? lp1

algebraicSort (lts:Split): Split ==
   lts := removeDuplicates lts
   sort(supDimElseRittWu?,lts)

moreAlgebraic?(ts:TS,us:TS): Boolean ==
   empty? ts => empty? us
   empty? us => true
   #ts < #us => false
   for p in (members us) repeat
      not algebraic?(mvar(p),ts) => return false
   true

subTriSet?(ts:TS,us:TS): Boolean ==
   empty? ts => true
   empty? us => false
   mvar(ts) > mvar(us) => false
   mvar(ts) < mvar(us) => subTriSet?(ts,rest(us)::TS)
   first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS,rest(us)::TS)
   false

internalSubPolSet?(lp1: LP, lp2: LP): Boolean ==
   empty? lp1 => true
   empty? lp2 => false
associates?((first lp1, first lp2) =>
  internalSubPolSet?(rest lp1, rest lp2)
infRittWu?(first lp1, first lp2) => false
internalSubPolSet?(lp1, rest lp2)

subPolSet?((lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1,lp2)

infRittWu?((lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1,lp2)

internalInfRittWu?((lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?((first lp1, first lp2)$P => true
  infRittWu?((second lp1, first lp2)$P => false
  infRittWu?((rest lp1, rest lp2)$

subCase? (lpwt1:LpWT,lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false

subQuasiComponent?(ts:TS,us:TS): Boolean ==
  -- "failed" is false iff saturate(us) is radical
  subTriSet?((us,ts) => true
  not moreAlgebraic?((ts,us) => false::Union(Boolean,"failed")
  for p in (members us) repeat
    mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
      return("failed":Union(Boolean,"failed"))
  for p in (members us) repeat
    not zero? initiallyReduce(p,ts) =>
      return("failed":Union(Boolean,"failed"))
  lsfp := squareFreeFactors(initials us)
  for p in lsfp repeat
    not invertible?(p,ts)@B =>
      return(false::Union(Boolean,"failed"))
  true::Union(Boolean,"failed")

subQuasiComponent?((ts:TS,us:TS): Boolean ==
  k: Key := [ts, us]
  e := extractIfCan(k)$H
  e case Entry => e::Entry
  ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts,us)
  b: Boolean := (ubf case Boolean) and (ubf::Boolean)
  insert!(k,b)$H
$b$

```
subQuasiComponent?(ts:TS,lus:Split): Boolean ==
  for us in lus repeat
    subQuasiComponent?(ts,us)@B => return true
  false

removeSuperfluousCases (cases:List LpWT) ==
  #cases < 2 => cases
  toSee :=
    sort((x:LpWT,y:LpWT):Boolean +->
      supDimElseRittWu?(x.tower,y.tower),cases)
  lpwt1,lpwt2 : LpWT
toSave,headmaxcases,maxcases,copymaxcases : List LpWT
  while not empty? toSee repeat
    lpwt1 := first toSee
    toSee := rest toSee
    toSave := []
    for lpwt2 in toSee repeat
      if subCase?(lpwt1,lpwt2)
        then
          lpwt1 := lpwt2
        else
          if not subCase?(lpwt2,lpwt1)
            then
              toSave := cons(lpwt2,toSave)
      if empty? maxcases
        then
          headmaxcases := [lpwt1]
          maxcases := headmaxcases
        else
          copymaxcases := maxcases
          while (not empty? copymaxcases) and _
            (not subCase?(lpwt1,first(copymaxcases))) repeat
            copymaxcases := rest copymaxcases
          if empty? copymaxcases
            then
              setrest!(headmaxcases,[lpwt1])
              headmaxcases := rest headmaxcases
          toSee := reverse toSave
    maxcases

removeSuperfluousQuasiComponents(lts: Split): Split ==
  lts := removeDuplicates lts
  #lts < 2 => lts
  toSee := algebraicSort lts
  toSave,headmaxlts,maxlts,copymaxlts : Split
  while not empty? toSee repeat
    ts := first toSee
    toSee := rest toSee
```
toSave := []
for us in toSee repeat
  if subQuasiComponent?(ts,us)@B
    then
      ts := us
    else
      if not subQuasiComponent?(us,ts)@B
        then
          toSave := cons(us,toSave)
  if empty? maxlts
    then
      headmaxlts := [ts]
      maxlts := headmaxlts
    else
      copymaxlts := maxlts
      while (not empty? copymaxlts) and _
        (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
        copymaxlts := rest copymaxlts
      if empty? copymaxlts
        then
          setrest!(headmaxlts,[ts])
          headmaxlts := rest headmaxlts
    toSee := reverse toSave
  algebraicSort maxlts

removeAssociates (lp:LP) := 
  removeDuplicates [primitivePart(p) for p in lp]

branchIfCan(leq: LP,ts: TS,lineq: LP, b1:B,b2:B,b3:B,b4:B,b5:B):UBF ==
  -- ASSUME pols in leq are squarefree and mainly primitive
  -- if b1 then CLEAN UP leq
  -- if b2 then CLEAN UP lineq
  -- if b3 then SEARCH for ZERO in lineq with leq
  -- if b4 then SEARCH for ZERO in lineq with ts
  -- if b5 then SEARCH for ONE in leq with lineq
  if b1
    then
      leq := removeAssociates(leq)
      leq := remove(zero?,leq)
      any?(ground?,leq) =>
        return("failed"::Union(Branch,"failed"))
  if b2
    then
      any?(zero?,lineq) =>
        return("failed"::Union(Branch,"failed"))
      lineq := removeRedundantFactors(lineq)$polsetpack
  if b3
    then
      ps: PS := construct(leq)$PS
      for q in lineq repeat
zero? remainder(q,ps).polnum =>
    return("failed"::Union(Branch,"failed"))

(empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch):UBF
if b4
    then
        for q in lineq repeat
            zero? initiallyReduce(q,ts) =>
                return("failed"::Union(Branch,"failed"))

if b5
    then
        newleq: LP := []
        for p in leq repeat
            for q in lineq repeat
                if mvar(p) = mvar(q)
                    then
                        g := gcd(p,q)
                        newp := (p exquo g)::P
                        ground? newp =>
                            return("failed"::Union(Branch,"failed"))
                        newleq := cons(newp,newleq)
                    else
                        newleq := cons(p,newleq)

        leq := newleq
        leq := sort(infRittWu?, removeDuplicates leq)
        ([leq, ts, lineq]$Branch):UBF

prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
    -- if b1 then REMOVE REDUNDANT COMPONENTS in lts
    -- if b2 then SPLIT the input system with squareFree
    lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))
    any?(ground?,lp) => []
    empty? lts => []
    if b1 then lts := removeSuperfluousQuasiComponents lts
    not b2 =>
        [[lp,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
    toSee: List Branch
    lq: LP := []
    toSee := [[lq,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
    empty? lp => toSee
    for p in lp repeat
        lsfp := squareFreeFactors(p)$polsetpack
        branches: List Branch := []
        lq := []
        for f in lsfp repeat
            for branch in toSee repeat
                leq : LP := branch.eq
                ts := branch.tower
                lineq : LP := branch.ineq
                ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true)@UBF
                ubf1 case "failed" => "leave"
ubf2: UBF :=
    branchIfCan([f],ts,lineq,false,false,true,true,true)@UBF
ubf2 case "failed" => "leave"
leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))
lineq :=
    sort(infRittWu?,removeDuplicates concat(ubf1.ineq,ubf2.ineq))
newBranch :=
    branchIfCan(leq,ts,lineq,false,false,false,false)
branches:= cons(newBranch::Branch,branches)
lq := cons(f,lq)
toSee := branches
sort((x,y) +-> supDimElseRittWu?(x.tower,y.tower),toSee)

---

package QFCAT2 QuotientFieldCategoryFunctions2

--- QuotientFieldCategoryFunctions2.input ---

)set break resume
)sys rm -f QuotientFieldCategoryFunctions2.output
)spool QuotientFieldCategoryFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show QuotientFieldCategoryFunctions2
--E 1

)spool
)lisp (bye)

--- QuotientFieldCategoryFunctions2.help ---
This package extends a function between integral domains to a mapping between their quotient fields.

See Also:
• )show QuotientFieldCategoryFunctions2

---

QuotientFieldCategoryFunctions2 (QFCAT2)

Exports:
map

---

)abbrev package QFCAT2 QuotientFieldCategoryFunctions2
++ Description:
++ This package extends a function between integral domains to a mapping between their quotient fields.

QuotientFieldCategoryFunctions2(A, B, R, S): Exports == Impl where
A, B: IntegralDomain
R : QuotientFieldCategory(A)
S : QuotientFieldCategory(B)

Exports == with
map: (A -> B, R) -> S
++ map(func,frac) applies the function func to the numerator and denominator of frac.
Impl => add
    map(f, r) == f(numer r) / f(denom r)

-- QFCAT2.dotabb --
"QFCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QFCAT2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"QFCAT2" -> "PFECAT"

package QUATCT2 QuaternionCategoryFunctions2

-- QuaternionCategoryFunctions2.input --

)spool QuaternionCategoryFunctions2.output
)clear all

-- S 1 of 4
q := quatern(2/11,-8,3/4,1)
--R
  --R
  --R 2 3
  --R (1) -- - 8i + - j + k
  --R 11 4
  --R
  Type: Quaternion(Fraction(Integer))
--E 1

-- S 2 of 4
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer
--R
  --R Function declaration f : Fraction(Integer) -> Complex(Fraction(
  --R Integer)) has been added to workspace.
  Type: Void
--E 3

-- S 3 of 4
map(f,q)
```
--R Compiling function f with type Fraction(Integer) -> Complex(Fraction
--R (Integer))
--R
--R 2 3
--R (3) - - 8i + - j + k
--R 11 4
--R
--R Type: Quaternion(Complex(Fraction(Integer)))))
--E 3

--)show QuaternionCategoryFunctions2
--R
--R QuaternionCategoryFunctions2(QR: QuaternionCategory(R),R: CommutativeRing,QS: QuaternionCategory(S),S: Com
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for QUATCT2
--R
--R----------------------------------- Operations ---------------------------------
--R map : ((R -> S),QR) -> QS
--R
--E 4

)spool
)lisp (bye)

——— QuaternionCategoryFunctions2.help ———

====================================================================
QuaternionCategoryFunctions2 examples
====================================================================

QuaternionCategoryFunctions2 implements functions between two
quaternion domains. The function map is used by the system
interpreter to coerce between quaternion types.

The package QuaternionCategoryFunctions2 provides a function map
to convert an expression in Quaternion(R) to an expression in
Quaternion(S) using the function f.

q := quatern(2/11,-8,3/4,1)

2 3
-- - 8i + - j + k
11 4

Type: Quaternion Fraction Integer
```
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer

Function declaration f : Fraction Integer -> Complex Fraction Integer has been added to workspace.

Type: Void

map(f,q)

Compiling function f with type Fraction Integer -> Complex Fraction Integer

\[
\begin{align*}
\frac{2}{11} & \quad \frac{3}{4} \\
-8i & + - j + k
\end{align*}
\]

Type: Quaternion Complex Fraction Integer

See Also:

| see Also: | o )show QuaternionCategoryFunctions2 |

---

QuaternionCategoryFunctions2 (QUATCT2)

Exports:

map

--- package QUATCT2 QuaternionCategoryFunctions2 ---

)abbrev package QUATCT2 QuaternionCategoryFunctions2
++ Author: Robert S. Sutor
++ Date Created: 23 May 1990
++ Date Last Updated: 23 May 1990
++ Description:
++ \spadtype{QuaternionCategoryFunctions2} implements functions between
++ two quaternion domains. The function \spadfun{map} is used by
++ the system interpreter to coerce between quaternion types.

QuaternionCategoryFunctions2(QR,R,QS,S) : Exports ==
  Implementation where
  R : CommutativeRing
  S : CommutativeRing
  QR : QuaternionCategory R
  QS : QuaternionCategory S
  Exports == with
    map: (R -> S, QR) -> QS
      ++ map(f,u) maps f onto the component parts of the quaternion u.
      ++ to convert an expression in Quaternion(R) to Quaternion(S)
      ++
      +++ X f(a:FRAC(INT)):COMPLEX(FRAC(INT)) == a:COMPLEX(FRAC(INT))
      +++ X q:=quatern(2/11,-8,3/4,1)
      +++ X map(f,q)

  Implementation == add
    map(fn : R -> S, u : QR): QS ==
      quatern(fn real u, fn imagI u, fn imagJ u, fn imagK u)$QS

"QUATCT2": [color="#FF4488",href="bookvol10.4.pdf#nameddest=QUATCT2"]
"ACFS": [color="#4488FF",href="bookvol10.2.pdf#nameddest=QUATCAT"]
"QUATCT2" -> "QUATCAT"
Chapter 19

Chapter R

package REP RadicalEigenPackage

--- RadicalEigenPackage.input ---

)set break resume
)sys rm -f RadicalEigenPackage.output
)spool RadicalEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RadicalEigenPackage
--E 1

)spool
)lisp (bye)

---

--- RadicalEigenPackage.help ---

====================================================================
RadicalEigenPackage examples
====================================================================

Package for the computation of eigenvalues and eigenvectors. This package works for matrices with coefficients which are rational functions over the integers.
The eigenvalues and eigenvectors are expressed in terms of radicals.

See Also:
- )show Radical EigenPackage
- )show Fraction(Polynomial(Integer))
\textbf{PACKAGE REP RADICALEIGENPACKAGE} 3173

\begin{verbatim}
M ==> Matrix(F)
MRE ==> Matrix(RE)
ST ==> SuchThat(SE,P)
NNI ==> NonNegativeInteger

EigenForm ==> Record(eigval:Union(F,ST),eigmult:NNI,eigvec:List(M))
RadicalForm ==> Record(radval:RE,radmult:Integer,radvect:List(MRE))

C == with
  radicalEigenvectors : M -> List(RadicalForm)
  ++ radicalEigenvectors(m) computes
  ++ the eigenvalues and the corresponding eigenvectors of the
  ++ matrix m;
  ++ when possible, values are expressed in terms of radicals.

  radicalEigenvector : (RE,M) -> List(MRE)
  ++ radicalEigenvector(c,m) computes the eigenvector(s) of the
  ++ matrix m corresponding to the eigenvalue c;
  ++ when possible, values are
  ++ expressed in terms of radicals.

  radicalEigenvalues : M -> List RE
  ++ radicalEigenvalues(m) computes the eigenvalues of the matrix m;
  ++ when possible, the eigenvalues are expressed in terms of radicals.

  eigenMatrix : M -> Union(MRE,"failed")
  ++ eigenMatrix(m) returns the matrix b
  ++ such that \(b*m*(inverse b)\) is diagonal,
  ++ or "failed" if no such b exists.

  normalise : MRE -> MRE
  ++ normalise(v) returns the column
  ++ vector v
  ++ divided by its euclidean norm;
  ++ when possible, the vector v is expressed in terms of radicals.

  gramschmidt : List(MRE) -> List(MRE)
  ++ gramschmidt(lv) converts the list of column vectors lv into
  ++ a set of orthogonal column vectors
  ++ of euclidean length 1 using the Gram–Schmidt algorithm.

  orthonormalBasis : M -> List(MRE)
  ++ orthonormalBasis(m) returns the orthogonal matrix b such that
  ++ \(b*m*(inverse b)\) is diagonal.
  ++ Error: if m is not a symmetric matrix.

T == add
  PI ==> PositiveInteger
\end{verbatim}
RSP := RadicalSolvePackage R
import EigenPackage R

---- Local Functions ----
evalvect : (M,RE,SE) -> MRE
innerprod : (MRE,MRE) -> RE

---- eval a vector of F in a radical expression ----
evalvect(vect:M,alg:RE,x:SE) : MRE ==
n:=nrows vect
xx:=kernel(x)$Kernel(RE)
w:MRE:=zero(n,1)$MRE
for i in 1..n repeat
  v:=eval(vect(i,1) :: RE,xx,alg)
  setelt(w,i,1,v)
w
---- inner product ----
innerprod(v1:MRE,v2:MRE): RE == (((transpose v1)* v2)::MRE)(1,1)

---- normalization of a vector ----
normalise(v:MRE) : MRE ==
normv:RE := sqrt(innerprod(v,v))
normv = 0$RE => v
(1/normv)*v

---- Eigenvalues of the matrix A ----
radicalEigenvalues(A:M): List(RE) ==
x:SE :=new()$SE
pol:= characteristicPolynomial(A,x) :: F
radicalRoots(pol,x)$RSP

---- Eigenvectors belonging to a given eigenvalue ----
---- expressed in terms of radicals ----
radicalEigenvector(alpha:RE,A:M) : List(MRE) ==
n:=nrows A
B:MRE := zero(n,n)$MRE
for i in 1..n repeat
  for j in 1..n repeat B(i,j):=(A(i,j))::RE
  B(i,i):= B(i,i) - alpha
[v::MRE for v in nullSpace B]

---- eigenvectors and eigenvalues ----
radicalEigenvectors(A:M) : List(RadicalForm) ==
leig:List EigenForm := eigenvectors A
n:=nrows A
sln:List RadicalForm := empty()
veclist: List MRE
for eig in leig repeat
  eig.eigval case F =>
    veclist := empty()
for ll in eig.eigvec repeat
m:MRE:=zero(n,1)
for i in 1..n repeat m(i,1):=(ll(i,1))::RE
veclist:=cons(m,veclist)
sln:=cons([[(eig.eigval)::F::RE,eig.eigmult,veclist]]$RadicalForm,sln)
sym := eig.eigval :: ST
xx:= lhs sym
lval : List RE := radicalRoots((rhs sym) :: F ,xx)$RSP
for alg in lval repeat
nsl:=[alg,eig.eigmult, [evalvect(ep,alg,xx) for ep in eig.eigvec]]$RadicalForm
sln:=cons(nsl,sln)
sln
---- orthonormalization of a list of vectors ----
---- Grahm - Schmidt process ----
gramschmidt(lvect:List(MRE)) : List(MRE) ==
lvect=[] => []
v:=lvect.first
n := nrows v
RMR:=RectangularMatrix(n:PI,1,RE)
orth:List(MRE):=[(normalise v)]
for v in lvect.rest repeat
pol:=((v:RMR)-(+/
[innerprod(w,v)*w):RMR for w in orth])):MRE
orth:=cons(normalise pol,orth)
orth

---- The matrix of eigenvectors ----
eigenMatrix(A:M) : Union(MRE,"failed") ==
lef:List(MRE):=[:eiv.radvect for eiv in radicalEigenvectors(A)]
n:=nrows A
#lef <n => "failed"
d:MRE:=copy(lef.first)
for v in lef.rest repeat d:=(horizConcat(d,v))::MRE
d

---- orthogonal basis for a symmetric matrix ----
orthonormalBasis(A:M):List(MRE) ==
"symmetric?(A) => error "the matrix is not symmetric"
basis:List(MRE):=[]
lvec:List(MRE) := []
alglist:List(RadicalForm):=radicalEigenvectors(A)
n:=nrows A
for alterm in alglist repeat
if (lvec:=alterm.radvect)=[] then error "sorry "
if #(lvec)>1 then
lvec := gramschmidt(lvec)
basis := [:lvec, basis]
else basis := [normalise(lvec.first), :basis]
basis

package SOLVERAD RadicalSolvePackage

)set break resume
)sys rm -f RadicalSolvePackage.output
)spool RadicalSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 21
b: Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
--R
--R
--R 3
--R 3x + 7
--R (1) --------
--R 2
--R 5x - 13
--R Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 21
radicalSolve(b, x)
--R
--R
--R (2)
--R 3++++ 3+++++ ++++ ++ 3+++ 3+++++ ++++ ++ 3+++
radicalSolve(b)

radicalSolve(b=0,x)

radicalSolve(b=0)

radicalRoots(b,x)
```
--R 3++++ 3++++ 3++++ 3++++ 3++++ 3++++
--R \|7 \|7 \|7 \|3 - \|7 - \|7 - \|3 - \|7
--R (6) [-----------------,-----------------]
--R 3++
--R \|3 2\|3 2\|3
--R Type: List(Expression(Integer))
--E 6

--S 7 of 21
contractSolve(b=0,x)
--R
--R
--R 3++++ 3++++ 3++++ 3++++ 3++++
--R 3++++ (\|7 \|3 - 1)\%A (- \|7 \|3 - 1)\%A
--R (7) [\%A ,-----------------,-----------------] | [\%A= - ]
--R 2 2 3
--R Type: SuchThat(List(Expression(Integer)),List(Equation(Expression(Integer))))
--E 7

--S 8 of 21
contractSolve(b,x)
--R
--R
--R 3++++ 3++++ 3++++ 3++++
--R 3++++ (\|7 \|3 - 1)\%B (- \|7 \|3 - 1)\%B
--R (8) [\%B,-----------------,-----------------] | [\%B= - ]
--R 2 2 3
--R Type: SuchThat(List(Expression(Integer)),List(Equation(Expression(Integer))))
--E 8

--S 9 of 21
c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
--R
--R
--R 2
--R y + 4
--R (9) ------
--R y + 1
--R Type: Fraction(Polynomial(Integer))
--E 9

--S 10 of 21
radicalSolve([b,c],[x,y])
--R
--R
--R (10)
--R 3++++ 3++++
--R - \|7 \|7 \|3 - \|7
--R [x= ----------------,y= - 2\|7 ]
--R 3++
```
radicalSolve([b,c])

radicalSolve([b=0,c=0],[x,y])
--R
--R (12)
--R 3++++ ++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [[x= ---------------------------, y= -2|\-1 ],
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= -2|\-1 ], [x= ------, y= -2|\-1 ],
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= 2|\-1 ], [x= ------, y= 2|\-1 ]
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= 2|\-1 ], [x= ------, y= 2|\-1 ]
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= 2|\-1 ], [x= ------, y= 2|\-1 ]
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= 2|\-1 ], [x= ------, y= 2|\-1 ]
--R 3++
--R 2\|3
--R 3++++ ++++ 3++++
--R \-7 \-1 3 \-7 ++++
--R [x= ---------------------------, y= 2|\-1 ], [x= ------, y= 2|\-1 ]
radicalRoots([b,c],[x,y])

---

[-------------, - 2\|- 1 ],

3++

2\|- 1

2\|- 1

3+-+

2\|- 1

3+-+

2\|- 1

[------, 2\|- 1 ],

2\|- 1

Type: List(List(Expression(Integer)))

clear all

t1:=radicalSolve(x^3 + x^2 - 7 = 0, x)

---

x =

9\|1295 + 187\|3

(- 9\| 3 + 9) +

3|

\| 54\|3

(- 3\| 3 - 3) +
\begin{align*}
3 | \quad +\quad \\
\backslash & \quad 54 \mid 3 \\
\hline
3 | \quad +\quad \\
\backslash & \quad 54 \mid 3 \\
\hline
9 \mid 1295 + 187 \mid 3 \\
\hline
9 \mid 1295 + 187 \mid 3 \\
\hline
9 \mid 1295 + 187 \mid 3 \\
\hline
9 \mid 1295 + 187 \mid 3 \\
\hline
x = \quad \frac{3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3}{3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3 \quad + \quad 3 \mid + \quad \backslash \quad 54 \mid 3} \\
\hline
\text{Type: List(Equation(Expression(Integer)))}
\end{align*}
(2) -> t2 := rhs(t1.1)
--R
--R (2)
--R +------------------+2
--R | +----+ +-+
--R +---+ |9\|1295 + 187\|3
--R (- 9\|- 3 + 9) |------------------
--R 3| +-+
--R \| 54\|3
--R +
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (- 9\|- 3 + 9) |------------------ - 2
--R 3| +-+
--R \| 54\|3
--R /
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 + 9) |------------------
--R 3| +-+
--R \| 54\|3
--R /
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 + 9) |------------------ + 2
--R 3| +-+
--R \| 54\|3
--R /
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 + 9) |------------------
--R 3| +-+
--R \| 54\|3

Type: Expression(Integer)

--E 16

--S 17 of 21
(3) -> t3 := rhs(t1.2)
--R
--R (3)
--R +------------------+2
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (- 9\|- 3 - 9) |------------------
--R 3| +-+
--R \| 54\|3
--R +
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 - 3) |------------------ - 2
--R 3| +-+
--R \| 54\|3
--R /
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 - 3) |------------------ + 2
--R 3| +-+
--R \| 54\|3
--R /
--R +------------------+
--R | +----+ +-+
--R +----- |9\|1295 + 187\|3
--R (9\|- 3 - 9) |------------------
--R 3| +-+
--R \| 54\|3
--R

---S 18 of 21
(4) -> t4:=rhs(t1.3)
---R

---S 19 of 21
(5) -> t2^3+t2^2-7
---R

---S 20 of 21
(6) -> t3^3+t3^2-7
---R

---S 21 of 21
(7) -> t4^3+t4^2-7
---R

)spool

| RadicalSolvePackage.help |

---E 17

---E 18

---E 19

---E 20

---E 21
RadicalSolvePackage examples

This package tries to find solutions expressed in terms of radicals for systems of equations of rational functions with coefficients in an integral domain \(R\).

\[
b: \text{Fraction(Polynomial(Integer))}:=(3x^3+7)/(5x^2-13)
\]

\[
3x + 7
\]

\[
---------
\]

\[
2
\]

\[
5x - 13
\]

\[
\text{radicalSolve}(b,x)
\]

\[
3^{+}++3^{+}+++3^{+}+++3^{+}+++3^{+}+++3^{+}+++\]

\[
\text{radicalSolve}(b)
\]

\[
\text{radicalSolve}(b=0,x)
\]

\[
\text{radicalSolve}(b=0)
\]

\[
\text{radicalRoots}(b,x)
\]
\[ y + 4 \]
\[ y + 1 \]

radicalSolve([b,c],[x,y])
\[
\begin{align*}
0 & \quad 2 | 3 \\
\end{align*}
\]
radicalSolve([b,c])

\[
\begin{align*}
&3 \quad \text{++++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
&\quad \text{++++} \\
See Also:
  o )show RadicalSolvePackage
RadicalSolvePackage (SOLVERAD)

Exports:
contractSolve radicalRoots radicalSolve
radicalSolve : (RF, SY) -> L EQ RE
++radicalSolve(rf,x) finds the solutions expressed in terms of
++radicals of the equation rf = 0 with respect to the symbol x,
++where rf is a rational function.
++
++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
++X radicalSolve(b,x)

radicalSolve : RF -> L EQ RE
++radicalSolve(rf) finds the solutions expressed in terms of
++radicals of the equation rf = 0, where rf is a
++univariate rational function.
++
++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
++X radicalSolve(b)

radicalSolve : (EQ RF, SY) -> L EQ RE
++radicalSolve(eq,x) finds the solutions expressed in terms of
++radicals of the equation of rational functions eq
++with respect to the symbol x.
++
++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
++X radicalSolve(b=0,x)

radicalSolve : EQ RF -> L EQ RE
++radicalSolve(eq) finds the solutions expressed in terms of
++radicals of the equation of rational functions eq
++with respect to the unique symbol x appearing in eq.
++
++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
++X radicalSolve(b=0)

radicalSolve : (L RF, L SY) -> L L EQ RE
++radicalSolve(lrf,lvar) finds the solutions expressed in terms of
++radicals of the system of equations lrf = 0 with
++respect to the list of symbols lvar,
++where lrf is a list of rational functions.
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
++X radicalSolve([b,c],[x,y])

radicalSolve : L RF -> L L EQ RE
++radicalSolve(lrf) finds the solutions expressed in terms of
++radicals of the system of equations lrf = 0, where lrf is a
++system of univariate rational functions.
++
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
++X radicalSolve([b,c])

radicalSolve : (L EQ RF, L SY) -> L L EQ RE
++radicalSolve(leq, lvar) finds the solutions expressed in terms of
++radicals of the system of equations of rational functions leq
++with respect to the list of symbols lvar.
++
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
++X radicalSolve([b=0, c=0], [x, y])

radicalSolve : L EQ RF -> L L EQ RE
++radicalSolve(leq) finds the solutions expressed in terms of
++radicals of the system of equations of rational functions leq
++with respect to the unique symbol x appearing in leq.
++
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
++X radicalSolve([b=0, c=0])

radicalRoots : (RF, SY) -> L RE
++radicalRoots(rf, x) finds the roots expressed in terms of radicals
++of the rational function rf with respect to the symbol x.
++
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X radicalRoots(b, x)

radicalRoots : (L RF, L SY) -> L L RE
++radicalRoots(lrf, lvar) finds the roots expressed in terms of
++radicals of the list of rational functions lrf
++with respect to the list of symbols lvar.
++
++X b:Fraction(Polynomial(Integer)) := (3*x^3+7)/(5*x^2-13)
++X c:Fraction(Polynomial(Integer)) := (y^2+4)/(y+1)
++X radicalRoots([b, c], [x, y])

contractSolve: (EQ RF, SY) -> SU
++contractSolve(eq, x) finds the solutions expressed in terms of
++radicals of the equation of rational functions eq
++ with respect to the symbol x. The result contains new symbols for common subexpressions in order to reduce the size of the output.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X contractSolve(b=0,x)

contractSolve: (RF,SY) -> SU
++contractSolve(rf,x) finds the solutions expressed in terms of radicals of the equation rf = 0 with respect to the symbol x, where rf is a rational function. The result contains new symbols for common subexpressions in order to reduce the size of the output.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X contractSolve(b,x)

Capsule => add
import DegreeReductionPackage(PR, R)
import SOLVEFOR

SideEquations: List EQ RE := []
ContractSoln: B := false

---- Local Function Declarations ----
solveInner:(PR, SY, B) -> SU
linear: UP -> List RE
quadratic: UP -> List RE
cubic: UP -> List RE
quartic: UP -> List RE
rad: PI -> RE
wrap: RE -> RE
New: RE -> RE
makeEq : (List RE,L SY) -> L EQ RE
select : L L RE -> L L RE
isGeneric?: (L PR,L SY) -> Boolean
findGenZeros : (L PR,L SY) -> L L RE
findZeros : (L PR,L SY) -> L L RE

New s ==
  s = 0 => 0
  S := new()$Symbol ::PR::RF::RE
  SideEquations := append([S = s], SideEquations)
  S

linear u == [(-coefficient(u,0))::RE /(coefficient(u,1))::RE]
quadratic u == quadratic(map(coerce,u)$UPF2)$SOLVEFOR
cubic u == cubic(map(coerce,u)$UPF2)$SOLVEFOR
quartic u == quartic(map(coerce,u)$UPF2)$SOLVEFOR
PACKAGE SOLVERAD RADICALSOLVE

rad n == n::Z::RE
wrap s == (ContractSoln => New s; s)

---- Exported Functions ----

-- find the zeros of components in "generic" position --
findGenZeros(rlp:L PR,rlv:L SY) : L L RE ==
  pp:=rlp.first
  v:=first rlv
  rlv:=rest rlv
  res:L L RE:=[]
  res:=append([reverse cons(r,[eval((-coefficient(univariate(p,vv),0)::RE)/
    (leadingCoefficient univariate(p,vv))::RE,
    kernel(v)@Kernel(RE),r) for vv in rlv for p in rlp.rest])
    for r in radicalRoots(pp::RF,v)],res)
  res

findZeros(rlp:L PR,rlv:L SY) : L L RE ==
  parRes:=[radicalRoots(p::RF,v) for p in rlp for v in rlv]
  parRes:=select parRes
  res:L L RE :=[]
  res1:L RE
  for par in parRes repeat
    res1:=[par.first]
    lv1:L Kernel(RE):=[kernel rlv.first]
    rlv1:=rlv.rest
    p1:=par.rest
    while p1^=[] repeat
      res1:=cons(eval(p1.first,lv1,res1),res1)
      p1:=p1.rest
      lv1:=cons(kernel rlv1.first,lv1)
      rlv1:=rlv1.rest
    res:=cons(res1,res)
  res

radicalSolve(pol:RF,v:SY) ==
  [equation(v::RE,r) for r in radicalRoots(pol,v)]

radicalSolve(p:RF) ==
  zero? p =>
    error "equation is always satisfied"
  lv:=removeDuplicates
  concat(variables numer p, variables denom p)
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  radicalSolve(p,lv.first)
CHAPTER 19. CHAPTER R

radicalSolve(eq: EQ RF) ==
    radicalSolve(lhs eq - rhs eq)

radicalSolve(eq: EQ RF, v: SY) ==
    radicalSolve(lhs eq - rhs eq, v)

radicalRoots(lp: L RF, lv: L SY) ==
    parRes := triangularSystems(lp, lv)$SystemSolvePackage(R)
    parRes = list [] => []
    -- select the components in "generic" form
    rlv := reverse lv
    rpRes := [reverse res for res in parRes]
    listGen := [res for res in rpRes | isGeneric?(res, rlv)]
    result := L L RE := []
    if listGen = [] then
        result := "append" /[findGenZeros(res, rlv) for res in listGen]
        for res in listGen repeat
            rpRes := delete(rpRes, position(res, rpRes))
        -- non-generic components
        rpRes = [] => result
        append("append" /[findZeros(res, rlv) for res in rpRes], result)

radicalSolve(lp: L RF, lv: L SY) ==
    [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(lp: L RF) ==
    lv := "setUnion" /[setUnion(variables numer p, variables denom p) for p in lp]
    [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(le: L EQ RF, lv: L SY) ==
    lp := [rhs p - lhs p for p in le]
    [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(le: L EQ RF) ==
    lp := [rhs p - lhs p for p in le]
    lv := "setUnion" /[setUnion(variables numer p, variables denom p) for p in lp]
    [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

contractSolve(eq: EQ RF, v: SY) ==
    solveInner(numer(lhs eq - rhs eq), v, true)

contractSolve(pq: RF, v: SY) == solveInner(numer pq, v, true)

radicalRoots(pq: RF, v: SY) == lhs solveInner(numer pq, v, false)
-- test if the ideal is radical in generic position --

isGeneric?(rlp:L PR, rlv:L SY) : Boolean ==
  "and"/[degree(f,x)=1 for f in rest rlp for x in rest rlv]

---- select the univariate factors

select(lp:L L RE) : L L RE ==
lp=[] => list []
[:[cons(f, lsel) for lsel in select lp.rest] for f in lp.first]

---- Local Functions ----

-- construct the equation

def makeEq(nres:L RE, lv:L SY) : L EQ RE ==
  [equation(x :: RE, r) for x in lv for r in nres]

solveInner(pq:PR, v:SY, contractFlag:B) ==
  SideEquations := []
  ContractSoln := contractFlag

  factors := factors
  (factor pq)$MultivariateFactorize(SY, IndexedExponents SY, R, PR)

  constants: List PR := []
  unsolved: List PR := []
  solutions: List RE := []

  for f in factors repeat
    ff := f.factor
    "member?(v, variables (ff)) =>
      constants := cons(ff, constants)
    "
    u := univariate(ff, v)
    t := reduce u
    u := t.pol
    n := degree u
    l: List RE :=
      n = 1 => linear u
      n = 2 => quadratic u
      n = 3 => cubic u
      n = 4 => quartic u
      unsolved := cons(ff, unsolved)

    for s in l repeat
      if t.deg > 1 then s := wrap s
      T0 := expand(s, t.deg)
      for i in 1..f.exponent repeat
        solutions := append(T0, solutions)
      re := SideEquations
      [solutions, SideEquations]$SU
package RADUTIL RadixUtilities

--- RadixUtilities.input ---

)set break resume
)sys rm -f RadixUtilities.output
)spool RadixUtilities.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show RadixUtilities
--E 1

)spool
)lisp (bye)

--- RadixUtilities.help ---

====================================================================
RadixUtilities examples
====================================================================

This package provides tools for creating radix expansions.

See Also:
  o )show RadixUtilities
RadixUtilities (RADUTIL)

Exports:
radix

--- package RADUTIL RadixUtilities ---

)abbrev package RADUTIL RadixUtilities
++ Author: Stephen M. Watt
++ Date Created: October 1986
++ Date Last Updated: May 15, 1991
++ Description:
++ This package provides tools for creating radix expansions.

RadixUtilities: Exports == Implementation where
Exports ==> with
    radix: (Fraction Integer,Integer) -> Any
    ++ radix(x,b) converts x to a radix expansion in base b.
Implementation ==> add
    radix(q, b) ==
        coerce(q :: RadixExpansion(b))$AnyFunctions1(RadixExpansion b)

--- RADUTIL.dotabb ---

"RADUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RADUTIL"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"RADUTIL" -> "PID"
"RADUTIL" -> "OAGROUP"
package RDIST RandomDistributions

--- RandomDistributions.input ---

)set break resume
)sys rm -f RandomDistributions.output
)spool RandomDistributions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RandomDistributions
--E 1

)spool
)lisp (bye)

---

--- RandomDistributions.help ---

====================================================================
RandomDistributions examples
====================================================================

This package exports random distributions

See Also:
  o )show RandomDistributions

---
RandomDistributions (RDIST)

Exports:
uniformd rdHack1  weighted

)abbrev package RDIST RandomDistributions
++ Description:
++ This package exports random distributions

RandomDistributions(S: SetCategory): with
  uniform: Set S -> (() -> S)
    ++ uniform(s) undocumented
  weighted: List Record(value: S, weight: Integer) -> (() -> S)
    ++ weighted(l) undocumented
  rdHack1: (Vector S, Vector Integer, Integer) -> (() -> S)
    ++ rdHack1(v, u, n) undocumented
==
import RandomNumberSource()

weighted lvw ==
  -- Collapse duplicates, adding weights.
  t: Table(S, Integer) := table()
  for r in lvw repeat
    u := search(r.value, t)
    w := (u case "failed" => 0; u::Integer)
    t r.value := w + r.weight

  -- Construct vectors of values and cumulative weights.
  kl := keys t
  n := (#kl)::NonNegativeInteger
  n = 0 => error "Cannot select from empty set"
  kv: Vector(S) := new(n, kl.0)
  vw: Vector(Integer) := new(n, 0)

  totwt: Integer := 0
for k in kl for i in 1..n repeat
  kv.i := k
  totwt := totwt + t k
  wv.i := totwt

-- Function to generate an integer and lookup.
rdHack1(kv, wv, totwt)

rdHack1(kv, wv, totwt) ==
  w := randnum totwt
  -- do binary search in wv
  kv.1

uniform fset ==
  l := members fset
  n := #l
  l.(randnum(n)+1)

package RFDIST RandomFloatDistributions

— RandomFloatDistributions.input —

)set break resume
)sys rm -f RandomFloatDistributions.output
)spool RandomFloatDistributions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RandomFloatDistributions
--E 1

)spool
)lisp (bye)
This package exports random floating-point distributions

See Also:
- )show RandomFloatDistributions

---

RandomFloatDistributions (RFDIST)

Exports:
- Beta
- chiSquare
- chiSquare1
- exponential
- exponential1
- F
- normal
- t
- normal01
- uniform
- uniform01

---

)abbrev package RFDIST RandomFloatDistributions
++ Description:
++ This package exports random floating-point distributions

RationalNumber==> Fraction Integer
RandomFloatDistributions(): Cat == Body where
  NNI ==> NonNegativeInteger
  Cat ==> with

  ...
uniform01: () -> Float
  ++ uniform01() undocumented
normal01: () -> Float
  ++ normal01() undocumented
exponential1: () -> Float
  ++ exponential1() undocumented
chiSquare1: NNI -> Float
  ++ chiSquare1(n) undocumented

uniform: (Float, Float) -> (() -> Float)
  ++ uniform(f,g) undocumented
normal: (Float, Float) -> (() -> Float)
  ++ normal(f,g) undocumented
exponential: (Float) -> (() -> Float)
  ++ exponential(f) undocumented
chiSquare: (NNI) -> (() -> Float)
  ++ chiSquare(n) undocumented
Beta: (NNI, NNI) -> (() -> Float)
  ++ Beta(n,m) undocumented
F: (NNI, NNI) -> (() -> Float)
  ++ F(n,m) undocumented
t: (NNI) -> (() -> Float)
  ++ t(n) undocumented

Body ==> add
import RandomNumberSource()
-- FloatPackage0()

-- random() generates numbers in 0..rnmax
rnmax := (size()$RandomNumberSource() - 1)::Float

uniform01() ==
  randnum()::Float/rnmax
uniform(a,b) ==
  a + uniform01()*(b-a)

exponential1() ==
  u: Float := 0
  -- This test should really be  u < m where m is
  -- the minumum acceptable argument to log.
  while u = 0 repeat u := uniform01() - log u
exponential(mean) ==
  mean*exponential1()

-- This method is correct but slow.
normal01() ==
  s := 2::Float
  while s >= 1 repeat
v1 := 2 * uniform01() - 1
v2 := 2 * uniform01() - 1
s := v1**2 + v2**2
v1 * sqrt(-2 * log s/s)

normal(mean, stdev) ==
mean + stdev*normal01()

chiSquare1 dgfree ==
x: Float := 0
for i in 1..dgfree quo 2 repeat
  x := x + 2*exponential1()
if odd? dgfree then
  x := x + normal01()**2
x

chiSquare dgfree ==
chiSquare1 dgfree

Beta(dgfree1, dgfree2) ==
y1 := chiSquare1 dgfree1
y2 := chiSquare1 dgfree2
y1/(y1 + y2)

F(dgfree1, dgfree2) ==
y1 := chiSquare1 dgfree1
y2 := chiSquare1 dgfree2
(dgfree2 * y1)/(dgfree1 * y2)

t dgfree ==
n := normal01()
\(d := \frac{\chi^2_{\text{dgfree}}}{(\text{dgfree}::\text{Float})}\)
n / sqrt d

— RFDIST.dotabb —

"RFDIST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFDIST"]
"Package" [color="#FF4488"]
"RFDIST" -> "Package"

package RIDIST RandomIntegerDistributions

— RandomIntegerDistributions.input —
```lisp
(set break resume)
(sys rm -f RandomIntegerDistributions.output)
(spool RandomIntegerDistributions.output)
(set message test on)
(set message auto off)
clear all

---S 1 of 1
(show RandomIntegerDistributions)
---E 1

(spool)
lisp (bye)
```

---

--- RandomIntegerDistributions.help ---

====================================================================
RandomIntegerDistributions examples
====================================================================

This package exports integer distributions

See Also:
  o )show RandomIntegerDistributions

---

RandomIntegerDistributions (RIDIST)

Exports:
  binomial geometric poisson ridHack1 uniform
package RIDIST RandomIntegerDistributions

++ Description:
++ This package exports integer distributions

RandomIntegerDistributions(): with
uniform: Segment Integer -> (() -> Integer)
  ++ uniform(s) as
  ++   l + u0 + w*u1 + w**2*u2 +...+ w**(n-1)*u-1 + w**n*m
  ++ where
  ++   s = a..b
  ++   l = min(a,b)
  ++   m = abs(b-a) + 1
  ++   w**n < m < w**(n+1)
  ++   u0,...,un-1 are uniform on 0..w-1
  ++   m is uniform on 0..(m quo w**n)-1
binomial: (Integer, RationalNumber) -> (() -> Integer)
  ++ binomial(n,f) \undocumented
poisson: RationalNumber -> (() -> Integer)
  ++ poisson(f) \undocumented
geometric: RationalNumber -> (() -> Integer)
  ++ geometric(f) \undocumented
ridHack1: (Integer,Integer,Integer,Integer) -> Integer
  ++ ridHack1(i,j,k,l) \undocumented

== add
import RandomNumberSource()
import IntegerBits()

uniform aTob ==
a := lo aTob; b := hi aTob
l := min(a,b); m := abs(a-b) + 1
w := 2**(bitLength size() quo 2)::NonNegativeInteger
n := 0
mq := m -- m quo w**n
while (mqnext := mq quo w) > 0 repeat
  n := n + 1
  mq := mqnext
  ridHack1(mq, n, w, l)

ridHack1(mq, n, w, l) ==
r := randnum mq
for i in 1..n repeat r := r*w + randnum w
r + 1

---
package RANDSRC RandomNumberSource

>set break resume
>sys rm -f RandomNumberSource.output
>spool RandomNumberSource.output
>set message test on
>set message auto off
>clear all

--S 1 of 1
>show RandomNumberSource
--E 1

>spool
>lisp (bye)

--

RandomNumberSource examples

All random numbers used in the system should originate from the same generator. This package is intended to be the source.

See Also:
  o >show RandomNumberSource

-----
RandomNumberSource (RANDSRC)

Exports:
randnum  reseed  seed  size

-- package RANDSRC RandomNumberSource --

)abbrev package RANDSRC RandomNumberSource
++ Author:S.M.Watt
++ Date Created: April 87
++ Date Last Updated: Jan 92, May 1995 (MCD)
++ Description:
++ Random number generators.
++ All random numbers used in the system should originate from
++ the same generator. This package is intended to be the source.
--
-- Possible improvements:
-- 1) Start where the user left off
-- 2) Be able to switch between methods in the random number source.

RandomNumberSource(): with

-- If r := randnum() then 0 <= r < size().
randnum: () -> Integer
  ++ randnum() is a random number between 0 and size().
-- If r := randnum() then 0 <= r < size().
size: () -> Integer
  ++ size() is the base of the random number generator

-- If r := randnum n and n <= size() then 0 <= r < n.
randnum: Integer -> Integer
  ++ randnum(n) is a random number between 0 and n.
reseed: Integer -> Void
  ++ reseed(n) restarts the random number generator at n.
seed : () -> Integer
  ++ seed() returns the current seed value.

== add
-- This random number generator passes the spectral test
-- with flying colours. [Knuth vol2, 2nd ed, p105]
ranbase: Integer := 2**31-1
x0: Integer := 1231231231
x1: Integer := 3243232987

randnum() ==
t := (271828183 * x1 - 314159269 * x0) rem ranbase
if t < 0 then t := t + ranbase
x0:= x1
x1:= t

size() == ranbase
reseed n ==
x0 := n rem ranbase
-- x1 := (n quo ranbase) rem ranbase
x1 := n quo ranbase

seed() == x1*ranbase + x0

-- Compute an integer in 0..n-1.
randnum n ==
(n * randnum()) quo ranbase

package RATFACT RationalFactorize

package RATFACT RationalFactorize
RationalFactorize help

RationalFactorize examples

Factorization of extended polynomials with rational coefficients. This package implements factorization of extended polynomials whose coefficients are rational numbers. It does this by taking the \text{lcm} of the coefficients of the polynomial and creating a polynomial with integer coefficients. The algorithm in \text{GaloisGroupFactorizer} is then used to factor the integer polynomial. The result is normalized with respect to the original \text{lcm} of the denominators.

See Also:
o \text{show RationalFactorize}

Exports:
factor factorSquareFree
package RATFACT RationalFactorize

)abbrev package RATFACT RationalFactorize
++ Author: P. Gianni
++ Date last updated: December 1993
++ Description:
++ Factorization of extended polynomials with rational coefficients.
++ This package implements factorization of extended polynomials
++ whose coefficients are rational numbers. It does this by taking the
++ lcm of the coefficients of the polynomial and creating a polynomial
++ with integer coefficients. The algorithm in
++ \spadtype{GaloisGroupFactorizer} is then
++ used to factor the integer polynomial. The result is normalized
++ with respect to the original lcm of the denominators.

RationalFactorize(RP) : public == private where
  I ==> Integer
  RN ==> Fraction Integer
  BP ==> SparseUnivariatePolynomial(I)
  RP : UnivariatePolynomialCategory RN

public ==> with

  factor : RP -> Factored RP
  ++ factor(p) factors an extended polynomial p over the rational numbers.
  factorSquareFree : RP -> Factored RP
  ++ factorSquareFree(p) factors an extended squareFree
  ++ polynomial p over the rational numbers.

private ==> add

import GaloisGroupFactorizer (BP)
ParFact ==> Record(irr:BP,pow:I)
FinalFact ==> Record(contp:I,factors:List(ParFact))
URNI ==> UnivariatePolynomialCategoryFunctions2(RN,RP,I,BP)
UIRN ==> UnivariatePolynomialCategoryFunctions2(I,BP,RN,RP)
fUnion ==> Union(“nil”, “sqfr”, “irred”, ”prime”)
FFE ==> Record(flg:fUnion, fctr:RP, xpnt:I)

factor(p:RP) : Factored(RP) ==
  p = 0 => 0
  pden: I := lcm([denom c for c in coefficients p])
  pol : RP := pden*p
  ipol: BP := map(numer,pol)$URNI
  ffact: FinalFact := henselFact(ipol,false)
  makeFR(((ffact.contp)/pden)::RP,
       [["prime",map(coerce,u.irr)$UIRN,u.pow]$FFE
       for u in ffact.factors])
factorSquareFree(p: RP) : Factored(RP) ==
  p = 0 => 0
  pden: I := lcm([denom c for c in coefficients p])
  pol : RP := pden*p
  ipol: BP := map(numer, pol)$URNI
  ffact: FinalFact := henselFact(ipol, true)
  makeFR(((ffact.contp)/pden)::RP,
    [["prime", map(coerce, u.irr)$UIRN, u.pow]$FFE
     for u in ffact.factors])}
RationalFunction examples
====================================================================
Utilities that provide the same top-level manipulations on fractions
than on polynomials.

See Also:
o )show RationalFunction

---

RationalFunction (RF)

Exports:
coerce eval mainVariable multivariate univariate variables

— package RF RationalFunction —

)abbrev package RF RationalFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 18 April 1991
++ Description:
++ Utilities that provide the same top-level manipulations on
++ fractions than on polynomials.
-- Do not make into a domain!

RationalFunction(R:IntegralDomain):Exports == Implementation where
V ==> Symbol
P ==> Polynomial R
Q ==> Fraction P
QF ==> PolynomialCategoryQuotientFunctions(IndexedExponents Symbol, Symbol, R, P, Q)
Exports ==> with
   variables : Q -> List V
      ++ variables(f) returns the list of variables appearing
      ++ in the numerator or the denominator of f.
   mainVariable: Q -> Union(V, "failed")
      ++ mainVariable(f) returns the highest variable appearing
      ++ in the numerator or the denominator of f, "failed" if
      ++ f has no variables.
   univariate : (Q, V) -> Fraction SparseUnivariatePolynomial Q
      ++ univariate(f, v) returns f viewed as a univariate
      ++ rational function in v.
   multivariate: (Fraction SparseUnivariatePolynomial Q, V) -> Q
      ++ multivariate(f, v) applies both the numerator and
      ++ denominator of f to v.
   eval : (Q, V, Q) -> Q
      ++ eval(f, v, g) returns f with v replaced by g.
   eval : (Q, List V, List Q) -> Q
      ++ eval(f, [v1,...,vn], [g1,...,gn]) returns f with
      ++ each vi replaced by gi in parallel, i.e. vi's appearing
      ++ inside the gi's are not replaced.
   eval : (Q, Equation Q) -> Q
      ++ eval(f, v = g) returns f with v replaced by g.
      ++ Error: if v is not a symbol.
   eval : (Q, List Equation Q) -> Q
      ++ eval(f, [v1 = g1,...,vn = gn]) returns f with
      ++ each vi replaced by gi in parallel, i.e. vi's appearing
      ++ inside the gi's are not replaced.
      ++ Error: if any vi is not a symbol.
   coerce : R -> Q
      ++ coerce(r) returns r viewed as a rational function over R.

Implementation ==> add
   foo : (List V, List Q, V) -> Q
   peval: (P, List V, List Q) -> Q

   coerce(r:R):Q == r::P::Q
   variables f == variables(f)$QF
   mainVariable f == mainVariable(f)$QF
   univariate(f, x) == univariate(f, x)$QF
   multivariate(f, x) == multivariate(f, x)$QF
   eval(x:Q, s:V, y:Q) == eval(x, [s], [y])
   eval(x:Q, eq:Equation Q) == eval(x, [eq])
   foo(ls, lv, x) == match(ls, lv, x, x::Q)$ListToMap(V, Q)

   eval(x:Q, l:List Equation Q) ==
      eval(x, [retract(lhs eq)@V for eq in l]$List(V),
            [rhs eq for eq in l]$List(Q))

   eval(x:Q, ls:List V, lv:List Q) ==
peval(numer x, ls, lv) / peval(denom x, ls, lv)

peval(p, ls, lv) ==
  map(z1 +-> foo(ls, lv, z1), z2 +-> z2::Q,p)
  $PolynomialCategoryLifting(IndexedExponents V,V,R,P,Q)

package DEFINTRF RationalFunctionDefiniteIntegration

RationalFunctionDefiniteIntegration examples

Definite integration of rational functions.
RationalFunctionDefiniteIntegration provides functions to compute definite integrals of rational functions.

See Also:
o )show RationalFunctionDefiniteIntegration

---

RationalFunctionDefiniteIntegration (DEFINTRF)

Exports:
integrate

--- package DEFINTRF RationalFunctionDefiniteIntegration ---

)abbrev package DEFINTRF RationalFunctionDefiniteIntegration
++ Author: Manuel Bronstein
++ Date Created: 2 October 1989
++ Date Last Updated: 2 February 1993
++ Description:
++ Definite integration of rational functions.
++ \spadtype{RationalFunctionDefiniteIntegration} provides functions to
++ compute definite integrals of rational functions.

RationalFunctionDefiniteIntegration(R): Exports == Implementation where
R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
RetractableTo Integer, LinearlyExplicitRingOver Integer)

SE ==> Symbol
RF ==> Fraction Polynomial R
FE ==> Expression R
ORF ==> OrderedCompletion RF
OFE ==> OrderedCompletion FE
**CHAPTER 19. CHAPTER R**

\[ U \Rightarrow \text{Union}(f_1: \text{OFE}, f_2: \text{List OFE}, \text{fail:"failed"}, \text{pole:"potentialPole"}) \]

Exports \( \Rightarrow \) with

\[ \text{integrate: (RF, SegmentBinding OFE) } \Rightarrow U \]
\[ \text{++ integrate}(f, x = a..b) \text{ returns the integral of} \]
\[ \text{++ } \int \! f(x) \, dx \text{ from } a \text{ to } b. \]
\[ \text{++ Error: if } f \text{ has a pole for } x \text{ between } a \text{ and } b. \]

\[ \text{integrate: (RF, SegmentBinding OFE, String) } \Rightarrow U \]
\[ \text{++ integrate}(f, x = a..b, "\text{noPole"}) \text{ returns the} \]
\[ \text{++ integral of } \int \! f(x) \, dx \text{ from } a \text{ to } b. \]
\[ \text{++ If it is not possible to check whether } f \text{ has a pole for } x \]
\[ \text{++ between } a \text{ and } b \text{ (because of parameters), then this function} \]
\[ \text{++ will assume that } f \text{ has no such pole.} \]
\[ \text{++ Error: if } f \text{ has a pole for } x \text{ between } a \text{ and } b \text{ or} \]
\[ \text{++ if the last argument is not "\text{noPole"}.} \]

-- the following two are contained in the above, but they are for the
-- interpreter... DO NOT COMMENT OUT UNTIL THE INTERPRETER IS BETTER!

\[ \text{integrate: (RF, SegmentBinding ORF) } \Rightarrow U \]
\[ \text{++ integrate}(f, x = a..b) \text{ returns the integral of} \]
\[ \text{++ } \int \! f(x) \, dx \text{ from } a \text{ to } b. \]

\[ \text{integrate: (RF, SegmentBinding ORF, String) } \Rightarrow U \]
\[ \text{++ integrate}(f, x = a..b, "\text{noPole"}) \text{ returns the} \]
\[ \text{++ integral of } \int \! f(x) \, dx \text{ from } a \text{ to } b. \]
\[ \text{++ If it is not possible to check whether } f \text{ has a pole for } x \]
\[ \text{++ between } a \text{ and } b \text{ (because of parameters), then this function} \]
\[ \text{++ will assume that } f \text{ has no such pole.} \]
\[ \text{++ Error: if } f \text{ has a pole for } x \text{ between } a \text{ and } b \text{ or} \]
\[ \text{++ if the last argument is not "\text{noPole"}.} \]

Implementation \( \Rightarrow \) add

import DefiniteIntegrationTools(R, FE)
import IntegrationResultRFToFunction(R)
import OrderedCompletionFunctions2(RF, FE)

\[ \text{int : (RF, SE, OFE, OFE, Boolean) } \Rightarrow U \]
\[ \text{nopole: (RF, SE, OFE, OFE) } \Rightarrow U \]

\[ \text{integrate}(f: RF, s: \text{SegmentBinding OFE}) \Rightarrow \]
\[ \text{int}(f, \text{variable } s, \text{lo segment } s, \text{hi segment } s, \text{false}) \]

\[ \text{nopole}(f, x, a, b) \Rightarrow \]
\[ k := \text{kernel}(x) @ \text{Kernel}(\text{FE}) \]
\[ (u := \text{integrate}(f, x)) \text{ case } \text{FE} \Rightarrow \]
\[ (v := \text{computeInt}(k, u::\text{FE}, a, b, \text{true})) \text{ case "failed" } \Rightarrow ["\text{failed"]} \]
\[ [v::\text{OFE}] \]
\[ \text{ans} := \text{empty}() @ \text{List}(\text{OFE}) \]
\[ \text{for } g \text{ in } u::\text{List}(\text{FE}) \text{ repeat} \]
\[ (v := \text{computeInt}(k, g, a, b, \text{true})) \text{ case "failed" } \Rightarrow \text{return ["failed"]} \]
\[ \text{ans} := \text{concat}!(\text{ans}, [v::\text{OFE}]) \]
[ans]

\[
\text{integrate}(f:RF, s:SegmentBinding ORF) == \\
\text{int}(f, \text{variable } s, \text{map}(x \mapsto x::\text{FE}, \text{lo segment } s), \\
\text{map}(x \mapsto x::\text{FE}, \text{hi segment } s), \text{false})
\]

\[
\text{integrate}(f:RF, s:SegmentBinding ORF, str:String) == \\
\text{int}(f, \text{variable } s, \text{map}(x \mapsto x::\text{FE}, \text{lo segment } s), \\
\text{map}(x \mapsto x::\text{FE}, \text{hi segment } s), \text{ignore? str})
\]

\[
\text{integrate}(f:RF, s:SegmentBinding OFE, str:String) == \\
\text{int}(f, \text{variable } s, \text{lo segment } s, \text{hi segment } s, \text{ignore? str})
\]

\[
\text{int}(f, x, a, b, \text{ignore?}) == \\
a = b \Rightarrow [0::\text{OFE}] \\
(z := \text{checkForZero}(\text{denom } f, x, a, b, \text{true})) \text{ case } \text{"failed" } \Rightarrow \\
\text{ignore? } \Rightarrow \text{nopole}(f, x, a, b) \\
[\text{"potentialPole"]} \\
z::\text{Boolean } \Rightarrow \text{error } \text{"integrate: pole in path of integration"} \\
nopole(f, x, a, b)
\]

---

--- DEFINTRF.dotabb ---

"DEFINTRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEFINTRF"] 
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"] 
"DEFINTRF" → "ACFS"

---

package RFFACT RationalFunctionFactor

--- RationalFunctionFactor.input ---

)set break resume 
)sys rm -f RationalFunctionFactor.output 
)spool RationalFunctionFactor.output 
)set message test on 
)set message auto off 
)clear all 

-- S 1 of 1 
)show RationalFunctionFactor
RationalFunctionFactor (RFFACT)

Exports:
factor

— package RFFACT RationalFunctionFactor —

)abbrev package RFFACT RationalFunctionFactor
++ Author: Patrizia Gianni
++ Description:
++ Factorization of univariate polynomials with coefficients which
++ are rational functions with integer coefficients.

RationalFunctionFactorizer(UP): Exports == Implementation where
  UP: UnivariatePolynomialCategory Fraction Polynomial Integer

  SE ==> Symbol
  P ==> Polynomial Integer
  RF ==> Fraction P
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2

Exports ==> with
  factor: UP -> Factored UP
  ++ factor(p) returns a prime factorisation of p.

Implementation ==> add

likuniv: (P, SE, P) -> UP

dummy := new()$SE

likuniv(p, x, d) ==
  map(y +-> y/d, univariate(p, x))$UPCF2(P, SparseUnivariatePolynomial P, RF, UP)

factor p ==
  d := denom(q := elt(p,dummy::P :: RF))
  map(x +-> likuniv(x,dummy,d),
      factor(numer q)$MultivariateFactorize(SE, IndexedExponents SE,Integer,P))$FactoredFunctions2(P, UP)

---

— RFFACT.dotabb —

"RFFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFFACT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RFFACT" -> "ALIST"

---

package RFFACTOR RationalFunctionFactorizer

— RationalFunctionFactorizer.input —

)set break resume
RationalFunctionFactorizer contains the factor function (called factorFraction) which factors fractions of polynomials by factoring the numerator and denominator. Since any non zero fraction is a unit the usual factor operation will just return the original fraction.

See Also:
- )show RationalFunctionFactorizer
Exports:
factor

— package RFFACTOR RationalFunctionFactorizer —

)abbrev package RFFACTOR RationalFunctionFactorizer
++ Author: P. Gianni
++ Date Last Updated: March 1995
++ Description:
++ \spadtype{RationalFunctionFactorizer} contains the factor function
++ (called factorFraction) which factors fractions of polynomials by factoring
++ the numerator and denominator. Since any non zero fraction is a unit
++ the usual factor operation will just return the original fraction.

RationalFunctionFactorizer(R) : C == T
where
R : EuclideanDomain  -- R with factor for R[X]
P ==> Polynomial R
FP ==> Fraction P
SE ==> Symbol

C == with
factorFraction : FP -> Fraction Factored(P)
    ++ factorFraction(r) factors the numerator and the denominator of
    ++ the polynomial fraction r.

T == add
factorFraction(p:FP) : Fraction Factored(P) ==
    R is Fraction Integer =>
        MR:=MRationalFactorize(IndexedExponents SE,SE,
                        Integer,P)
        (factor(numer p)$MR)/ (factor(denom p)$MR)
    R has FiniteFieldCategory =>
        FF:=MultFiniteFactorize(SE, IndexedExponents SE, R, P)
        (factor(numer p))$FF/(factor(denom p))$FF
    R has CharacteristicZero =>
        MFF:=MultivariateFactorize(SE, IndexedExponents SE, R, P)
        (factor(numer p))$MFF/(factor(denom p))$MFF
    error "case not handled"

—— RFFACTOR.dotabb ——

"RFFACTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFFACTOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RFFACTOR" -> "PFECAT"

package INTRF RationalFunctionIntegration

---

RationalFunctionIntegration.input

)set break resume
)sys rm -f RationalFunctionIntegration.output
)spool RationalFunctionIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionIntegration
--E 1

)spool
)lisp (bye)

---

RationalFunctionIntegration.help

====================================================================
RationalFunctionIntegration examples
====================================================================

This package provides functions for the integration of rational functions.

See Also:
  o )show RationalFunctionIntegration

---
RationalFunctionIntegration (INTRF)

Exports:
extendedIntegrate infieldIntegrate internalIntegrate limitedIntegrate

— package INTRF RationalFunctionIntegration —

)abbrev package INTRF RationalFunctionIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 29 Mar 1990
++ Description:
++ This package provides functions for the integration of rational functions.

RationalFunctionIntegration(F): Exports == Implementation where
  F: Join(IntegralDomain, RetractableTo Integer, CharacteristicZero)

SE ==> Symbol
P ==> Polynomial F
Q ==> Fraction P
UP ==> SparseUnivariatePolynomial Q
QF ==> Fraction UP
LGQ ==> List Record(coeff:Q, logand:Q)
UQ ==> Union(Record(ratpart:Q, coeff:Q), "failed")
ULQ ==> Union(Record(mainpart:Q, limitedlogs:LGQ), "failed")

Exports ==> with
  internalIntegrate: (Q, SE) -> IntegrationResult Q
    ++ internalIntegrate(f, x) returns g such that \( \frac{dg}{dx} = f \).
  infieldIntegrate : (Q, SE) -> Union(Q, "failed")
    ++ infieldIntegrate(f, x) returns a fraction
    ++ g such that \( \frac{dg}{dx} = f \)
    ++ if g exists, "failed" otherwise.
  limitedIntegrate : (Q, SE, List Q) -> ULQ
    ++ \( \frac{\text{limitedIntegrate}(f, x, [g_1, \ldots, g_n])}{\text{return}} \)
    ++ \( \frac{\text{fractions}}{\text{such that the g_i's are among}}
    ++ \( \frac{[g_1, \ldots, g_n]}{,}\)
++ \spad{dci/dx = 0}, and \spad{d(h + \text{sum}(ci \log(gi)))/dx = f} 
++ if possible, "failed" otherwise.

extendedIntegrate: (Q, SE, Q) -> UQ
++ extendedIntegrate(f, x, g) returns fractions \spad{[h, c]} such that
++ \spad{dc/dx = 0} and \spad{dh/dx = f - cg}, if \spad{[h, c]} exist,
++ "failed" otherwise.

Implementation ==> add
import RationalIntegration(Q, UP)
import IntegrationResultFunctions2(QF, Q)
import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
SE, F, P, Q)

infieldIntegrate(f, x) ==
map(x1 +-> multivariate(x1, x), infieldint univariate(f, x))

internalIntegrate(f, x) ==
map(x1 +-> multivariate(x1, x), integrate univariate(f, x))

extendedIntegrate(f, x, g) ==
map(x1 +-> multivariate(x1, x),
extendedint(univariate(f, x), univariate(g, x)))

limitedIntegrate(f, x, lu) ==
map(x1 +-> multivariate(x1, x),
limitedint(univariate(f, x), [univariate(u, x) for u in lu]))

— INTRF.dotabb —

"INTRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTRF" -> "PFECAT"

— RationalFunctionLimitPackage.input —

)set break resume
)sys rm -f RationalFunctionLimitPackage.output
)spool RationalFunctionLimitPackage.output
RationalFunctionLimitPackage (LIMITRF)

Exports:
complexLimit  limit

--- package LIMITRF RationalFunctionLimitPackage ---

Computation of limits for rational functions.
See Also:
- )show RationalFunctionLimitPackage

---
\texttt{\textbackslash )abbrev package LIMITRF RationalFunctionLimitPackage}

\texttt{Author: Manuel Bronstein}

\texttt{Date Created: 4 October 1989}

\texttt{Date Last Updated: 26 November 1991}

\texttt{Description:}

\texttt{Computation of limits for rational functions.}

\texttt{RationalFunctionLimitPackage(R:GcdDomain):Exports==Implementation where}

\texttt{Z ==> Integer}

\texttt{P ==> Polynomial R}

\texttt{RF ==> Fraction P}

\texttt{EQ ==> Equation}

\texttt{ORF ==> OrderedCompletion RF}

\texttt{OPF ==> OnePointCompletion RF}

\texttt{UP ==> SparseUnivariatePolynomial RF}

\texttt{SE ==> Symbol}

\texttt{QF ==> Fraction SparseUnivariatePolynomial RF}

\texttt{Result ==> Union(ORF, "failed")}

\texttt{TwoSide ==> Record(leftHandLimit:Result, rightHandLimit:Result)}

\texttt{U ==> Union(ORF, TwoSide, "failed")}

\texttt{RFSGN ==> RationalFunctionSign(R)}

\texttt{Exports ==> with}

\texttt{limit: (RF,EQ ORF) -> U}

\texttt{++ limit(f(x),x,a) computes the real two-sided limit \(\lim(x \rightarrow a,f(x))\)}

\texttt{complexLimit: (RF,EQ OPF) -> OPF}

\texttt{++ \texttt{complexLimit(f(x),x,a)} computes the complex limit \(\lim(x \rightarrow a,f(x))\)}

\texttt{limit: (RF,EQ RF,String) -> Result}

\texttt{++ \texttt{limit(f(x),x,a,"left") computes the real limit}}

\texttt{++ of f as its argument x approaches \texttt{\textbackslash spad{\{a\}}} from the left;}

\texttt{++ \texttt{limit(f(x),x,a,"right") computes the corresponding limit as x}}

\texttt{++ approaches \texttt{\textbackslash spad{\{a\}}} from the right.}
Implementation ==> add
import ToolsForSign R
import InnerPolySign(RF, UP)
import RFSGN
import PolynomialCategoryQuotientFunctions(IndexedExponents SE, SE, R, P, RF)

finiteComplexLimit: (QF, RF) -> OPF
finiteLimit : (QF, RF) -> U
fLimit : (Z, UP, RF, Z) -> Result

-- These 2 should be exported, see comment above
locallimit : (RF, SE, ORF) -> U
locallimitcomplex: (RF, SE, OPF) -> OPF

limit(f:RF, eq:EQ RF) ==
(xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
x := xx :: SE; a := rhs eq
locallimit(f,x,a::ORF)

complexLimit(f:RF, eq:EQ RF) ==
(xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
x := xx :: SE; a := rhs eq
locallimitcomplex(f,x,a::OPF)

limit(f:RF, eq:EQ OrderedCompletion P) ==
(p := retractIfCan(lhs eq)@Union(P,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
(xx := retractIfCan(p)@Union(SE,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
x := xx :: SE
a := map(y +-> y::RF,rhs eq)$OrderedCompletionFunctions2(P,RF)
lcallimit(f,x,a)

complexLimit(f:RF, eq:EQ OnePointCompletion P) ==
(p := retractIfCan(lhs eq)@Union(P,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
(xx := retractIfCan(p)@Union(SE,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
x := xx :: SE
a := map(y +-> y::RF,rhs eq)$OnePointCompletionFunctions2(P,RF)
lcallimitcomplex(f,x,a)

fLimit(n, d, a, dir) ==
  (s := signAround(d, a, dir, sign$RFSGN)) case "failed" => "failed"
n * (s::Z) * plusInfinity()

finiteComplexLimit(f, a) ==
zero?(n := (numer f) a) => 0
zero?(d := (denom f) a) => infinity()
(n / d)::OPF

finiteLimit(f, a) ==
zero?(n := (numer f) a) => 0
zero?(d := (denom f) a) =>
(s := sign(n)$RFSGN) case "failed" => "failed"
rs1 := fLimit(s::Z, denom f, a, 1)
lhsl := fLimit(s::Z, denom f, a, -1)
rhsl case "failed" =>
  lhsl case "failed" => "failed"
[rhsl, lhsl]
lhsl case "failed" => [lhsl, rhsl]
rhsl::ORF = lhsl::ORF => lhsl::ORF
[rhsl, lhsl]
(n / d)::ORF

locallimit(f,x,a) ==
g := univariate(f, x)
zero?(n := whatInfinity a) => finiteLimit(g, retract a)
(dn := degree numer g) > (dd := degree denom g) =>
  (sn := signAround(numer g, n, sign$RFSGN)) case "failed" => "failed"
  (sd := signAround(denom g, n, sign$RFSGN)) case "failed" => "failed"
  (sn::Z) * (sd::Z) * plusInfinity()
  dn < dd => 0
  ((leadingCoefficient numer g) / (leadingCoefficient denom g))::ORF

limit(f,eq,st) ==
  (xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE; a := rhs eq
  zero?(n := (numer(g := univariate(f, x))) a) => 0
  zero?(d := (denom g) a) =>
    (s := sign(n)$RFSGN) case "failed" => "failed"
    fLimit(s::Z, denom g, a, direction st)
  (n / d)::ORF

locallimitcomplex(f,x,a) ==
g := univariate(f, x)
  (r := retractIfCan(a)@Union(RF,"failed")) case RF =>
    finiteComplexLimit(g, r::RF)
  (dn := degree numer g) > (dd := degree denom g) => infinity()
  dn < dd => 0
  ((leadingCoefficient numer g) / (leadingCoefficient denom g))::OPF


package SIGNRF RationalFunctionSign

--- RationalFunctionSign.input ---

)set break resume
)sys rm -f RationalFunctionSign.output
)spool RationalFunctionSign.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionSign
--E 1

)spool
)lisp (bye)

---

--- RationalFunctionSign.help ---

====================================================================
RationalFunctionSign examples
====================================================================

Find the sign of a rational function around a point or infinity.

See Also:
  o )show RationalFunctionSign

---
RationalFunctionSign (SIGNRF)

Exports:
sign

--- package SIGNRF RationalFunctionSign ---

)abbrev package SIGNRF RationalFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 23 August 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Find the sign of a rational function around a point or infinity.

RationalFunctionSign(R:GcdDomain): Exports == Implementation where
  SE ==> Symbol
  P ==> Polynomial R
  RF ==> Fraction P
  ORF ==> OrderedCompletion RF
  UP ==> SparseUnivariatePolynomial RF
  U ==> Union(Integer, "failed")
  SGN ==> ToolsForSign(R)

Exports ==> with
  sign: RF -> U
    ++ sign f returns the sign of f if it is constant everywhere.
  sign: (RF, SE, ORF) -> U
    ++ sign(f, x, a) returns the sign of f as x approaches \spad{a},
    ++ from both sides if \spad{a} is finite.
  sign: (RF, SE, RF, String) -> U
    ++ sign(f, x, a, s) returns the sign of f as x nears \spad{a} from
    ++ the left (below) if s is the string \spad{"left"},
    ++ or from the right (above) if s is the string \spad{"right"}.

Implementation ==> add
  import SGN
  import InnerPolySign(RF, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents SE, SE, R, P, RF)

psign : P -> U
sqfrSign : P -> U
termSign : P -> U
listSign : (List P, Integer) -> U
finiteSign: (Fraction UP, RF) -> U

sign f ==
  (un := psign numer f) case "failed" => "failed"
  (ud := psign denom f) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

finiteSign(g, a) ==
  (ud := signAround(denom g, a, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, a, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

sign(f, x, a) ==
  g := univariate(f, x)
  zero?(n := whatInfinity a) => finiteSign(g, retract a)
  (ud := signAround(denom g, n, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, n, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

sign(f, x, a, st) ==
  (ud := signAround(denom(g := univariate(f, x)), a, d := direction st, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, a, d, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

psign p ==
  (r := retractIfCan(p)@Union(R, "failed")) case R => sign(r::R)$SGN
  (u := sign(retract(unit(s := squareFree p))@R)$SGN) case "failed" => "failed"
  ans := u::Integer
  for term in factors s | odd?(term.exponent) repeat
    (u := sqfrSign(term.factor)) case "failed" => return "failed"
    ans := ans * (u::Integer)
  ans

sqfrSign p ==
  (u := termSign first(l := monomials p)) case "failed" => "failed"
  listSign(rest l, u::Integer)

listSign(l, s) ==
  for term in l repeat
    (u := termSign term) case "failed" => return "failed"
    u::Integer ^= s => return "failed"
termSign \( \text{term} \) ==

\[
\text{for var in variables term repeat } \\
\text{odd? degree(\text{term, var}) \to return "failed"} \\
\text{sign(leadingCoefficient term)$SGN}
\]

---

--- SIGNRF.dotabb ---

"SIGNRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIGNRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SIGNRF" \to "PFECAT"

---

package SUMRF RationalFunctionSum

--- RationalFunctionSum.input ---

)set break resume
)sys rm -f RationalFunctionSum.output
)spool RationalFunctionSum.output
)set message test on
)set message auto off
)clear all

--S 1 of 13
sum(i::Polynomial(Integer),variable(i=1..n))
--R
--R
--R 2
--R 6i - 6i + 1
--R (1) ---------
--R 12
--R Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 13
sum(i::Fraction(Polynomial(Integer)),i::Symbol)
--R
--R
--R 2
--R 6i - 6i + 1
--R (2) -----------
--R 12
--R Type: Union(Fraction(Polynomial(Integer)),...)
--E 2

--S 3 of 13
sum(i,i=1..n)
--R
--R
--R 2
--R n + n
--R (3) ------
--R 2
--R Type: Fraction(Polynomial(Integer))
--E 3

--S 4 of 13
sum(i::Fraction(Polynomial(Integer)),i=1..n)
--R
--R
--R 2
--R n + n
--R (4) ------
--R 2
--R Type: Union(Fraction(Polynomial(Integer)),...)
--E 4

--S 5 of 13
s:=i=1..n
--R
--R
--R (5) i = 1..n
--R Type: SegmentBinding(Polynomial(Integer))
--E 5

--S 6 of 13
hiseg:=high(segment(s))
--R
--R
--R (6) n
--R Type: Polynomial(Integer)
--E 6

--S 7 of 13
loseg:=low(segment(s))
--R
--R
--R (7) 1
--R Type: Polynomial(Integer)
v := variable s

\( i \)

Type: Symbol

p := i::Polynomial(Integer)

\( i \)

Type: Polynomial(Integer)

f := sum(p, v)

\( \frac{6i - 6i + 1}{12} \)

Type: Fraction(Polynomial(Integer))

t1 := eval(f, v, (1+hiseg))

\( \frac{6n + 6n + 1}{12} \)

Type: Fraction(Polynomial(Integer))

t2 := eval(f, v, loseg)

\( \frac{1}{12} \)

Type: Fraction(Polynomial(Integer))
RationalFunctionSum examples

Computes sums of rational functions.

There are 4 different forms of the sum operator in this domain. They are variations of the types of the input

```plaintext
sum(i::Polynomial(Integer),variable(i=1..n))

2
6i - 6i + 1
----------
12
```

```plaintext
sum(i::Fraction(Polynomial(Integer)),i::Symbol)

2
6i - 6i + 1
----------
12
```

```plaintext
sum(i,i=1..n)

2
n + n
-----
2
```
We can compute the sum form on part at a time to see what happens. First, we make a SegmentBinding(Polynomial(Integer)):

\[ s := \text{SegmentBinding}(i = 1..n) \]

We pick out the upper and lower bounds of the segment:

\[ \text{hiseg} := \text{high}(s) \]
\[ \text{loseg} := \text{low}(s) \]

We pick out the variable from the segment:

\[ v := \text{variable}(s) \]

We create a polynomial we wish to sum:

\[ p := i::\text{Polynomial(Integer)} \]

And we create the sum for that polynomial:

\[ f := \text{sum}(p, v) \]
The new evaluate it at the upper endpoint:

\[
t_1 := \text{eval}(f, v, (1+\text{hiseg}))
\]

\[
\frac{2}{6n + 6n + 1} \frac{\text{------------}}{12}
\]

And the lower endpoint:

\[
t_2 := \text{eval}(f, v, \text{loseg})
\]

\[
\frac{1}{12}
\]

And we take the difference of the endpoints:

\[
t_1 - t_2
\]

\[
\frac{2}{n + n} \frac{\text{------}}{2}
\]

See Also:

\[
\text{show RationalFunctionSum}
\]
RationalFunctionSum (SUMRF)

Exports:

\( \text{sum} \)

--- package SUMRF RationalFunctionSum ---

\text{RationalFunctionSum}(R): \text{Exports} = \text{Impl} \text{ where}
\begin{align*}
R & \text{ is Join(IntegralDomain, OrderedSet, RetractableTo Integer)} \\
\text{P} & \Rightarrow \text{Polynomial } R \\
\text{RF} & \Rightarrow \text{Fraction } \text{P} \\
\text{FE} & \Rightarrow \text{Expression } R \\
\text{SE} & \Rightarrow \text{Symbol}
\end{align*}

Exports \Rightarrow with

\begin{align*}
\text{sum: (P, SE)} & \Rightarrow \text{RF} \\
& \text{sum(a(n), n) returns \text{A} which} \\
& \text{is the indefinite sum of \text{a} with respect to} \\
& \text{upward difference on \text{n}, i.e. \text{A(n+1) - A(n) = a(n)}}. \\
& \text{sum: (RF, SE)} & \Rightarrow \text{Union(RF, FE)} \\
& \text{sum(a(n), n) returns \text{A} which} \\
& \text{is the indefinite sum of \text{a} with respect to} \\
& \text{upward difference on \text{n}, i.e. \text{A(n+1) - A(n) = a(n)}}. \\
& \text{sum: (P, SegmentBinding P)} & \Rightarrow \text{RF} \\
& \text{sum(f(n), n = a..b) returns \text{f(a) + f(a+1) + ... f(b)}}.
\end{align*}
++
++X sum(i,i=1..n)
sum: (RF, SegmentBinding RF) -> Union(RF, FE)
++ sum(f(n), n = a..b) returns \spad{f(a) + f(a+1) + \ldots f(b)}.
++
++X sum(i::Fraction(Polynomial(Integer)),i=1..n)

Impl => add
import RationalFunction R
import GosperSummationMethod(IndexedExponents SE, SE, R, P, RF)

innersum : (RF, SE) -> Union(RF, "failed")
innersum: (P, SE) -> RF

sum(f:RF, s:SegmentBinding RF) ==
  (indef := innersum(f, v := variable s)) case "failed" =>
  summation(f::FE,map((z:RF):FE +->z::FE,s)
    $SegmentBindingFunctions2(RF,FE))
  eval(indef::RF, v, 1 + hi segment s)
  - eval(indef::RF, v,lo segment s)

sum(an:RF, n:SE) ==
  (u := innersum(an, n)) case "failed" => summation(an::FE, n)
  u::RF

sum(p:P, s:SegmentBinding P) ==
  f := sum(p, v := variable s)
  eval(f, v, (1 + hi segment s)::RF) - eval(f,v,lo(segment s)::RF)

innersum(an, n) ==
  (r := retractIfCan(an)@Union(P, "failed")) case "failed" =>
  an1 := eval(an, n, -1 + n::RF)
  (u := GosperMethod(an/an1, n, new$SE)) case "failed" =>
    "failed"
  an1 * eval(u::RF, n, -1 + n::RF)
  sum(r::P, n)

sum(p:P, n:SE) ==
  rec := sum(p, n)$InnerPolySum(IndexedExponents SE, SE, R, P)
  rec.num / (rec.den :: P)

—— SUMRF.dotabb ——
package INTRAT RationalIntegration

---

RationalIntegration.input ---

)set break resume
)sy rm -f RationalIntegration.output
)spool RationalIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalIntegration
---E 1

)spool
)lisp (bye)

---

RationalIntegration.help ---

Rational function integration. This package provides functions for the base case of the Risch algorithm.

See Also:
- )show RationalIntegration
RationalIntegration (INTRAT)

Exports:
extendedint  infieldint  integrate  limitedint

--- package INTRAT RationalIntegration ---

)abbrev package INTRAT RationalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ Rational function integration
++ This package provides functions for the base case of the Risch algorithm.
-- Used internally bt the integration packages

RationalIntegration(F, UP): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP: UnivariatePolynomialCategory F

RF ==> Fraction UP
IR ==> IntegrationResult RF
LLG ==> List Record(coeff:RF, logand:RF)
URF ==> Union(Record(ratpart:RF, coeff:RF), "failed")
U ==> Union(Record(mainpart:RF, limitedlogs:LLG), "failed")

Exports ==> with
  integrate : RF -> IR
    ++ integrate(f) returns g such that \spad{g' = f}.
  infieldint : RF -> Union(RF, "failed")
    ++ infieldint(f) returns g such that \spad{g' = f} or "failed"
    ++ if the integral of f is not a rational function.
  extendedint : (RF, RF) -> URF
    ++ extendedint(f, g) returns fractions \spad{[h, c]} such that
    ++ \spad{c' = 0} and \spad{h' = f - cg},
    ++ if \spad{[h, c]} exist, "failed" otherwise.
  limitedint : (RF, List RF) -> U
\[\text{\texttt{limitedint}(f, \{g_1, \ldots, g_n\}) \text{ returns fractions } \text{\texttt{spad}}\{h, [\{ci, g_i\}]\}\]
++ such that the \(g_i\)'s are among \text{\texttt{spad}}\{g_1, \ldots, g_n\}, \text{\texttt{spad}}\{ci' = 0\}, and
++ \text{\texttt{spad}}\{(h+\sum(ci \log(g_i)))' = f\}, if possible, "failed" otherwise.

Implementation ==> add
import TranscendentalIntegration(F, UP)

infeldint f ==
rec := baseRDE(0, f)$TranscendentalRischDE(F, UP)
rec.nosol => "failed"
rec.ans

integrate f ==
rec := monomialIntegrate(f, differentiate)
integrate(rec.polypart)::RF::IR + rec.ir

limitedint(f, lu) ==
quorem := divide(numer f, denom f)
(u := primliminfraction(quorem.remainder / (denom f), differentiate,
lu)) case "failed" => "failed"
[u.mainpart + integrate(quorem.quotient)::RF, u.limitedlogs]

extendedint(f, g) ==
fqr := divide(numer f, denom f)
gqr := divide(numer g, denom g)
(i1 := primextinfraction(fqr.remainder / (denom f), differentiate,
gqr.remainder / (denom g))) case "failed" => "failed"
i2:=integrate(fqr.quotient-retract(i1.coeff)@UP *gqr.quotient)::RF
[i2 + i1.ratpart, i1.coeff]
package RINTERP RationalInterpolation

Introduction

This file contains a crude naïve implementation of rational interpolation, where the coefficients of the rational function are in any given field.

Questions and Outlook

- Maybe this file should be joined with pinterp.spad, where polynomial Lagrange interpolation is implemented. I have a second version that parallels the structure of pinterp.spad closely.
- There are probably better ways to implement rational interpolation. Maybe http://www.cs.ucsb.edu/omer/personal/abstracts/rational.html contains something useful, but I don’t know.
- Comments welcome!

— RationalInterpolation.input —

)set break resume
)sys rm -f RationalInterpolation.output
)spool RationalInterpolation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalInterpolation
--E 1

)spool
)lisp (bye)

— RationalInterpolation.help —

====================================================================
RationalInterpolation examples
====================================================================

This package exports rational interpolation algorithms

See Also:
  o )show RationalInterpolation
RationalInterpolation (RINTERP)

Exports:
interpolate

— package RINTERP RationalInterpolation —

)abbrev package RINTERP RationalInterpolation
++ Description:
++ This package exports rational interpolation algorithms

RationalInterpolation(xx,F): Exports == Implementation where
xx: Symbol
F: Field
Exports == with
   interpolate: (List F, List F, NonNegativeInteger,
               NonNegativeInteger) -> Fraction Polynomial F

The implementation sets up a system of linear equations and solves it.
— package RINTERP RationalInterpolation —

Implementation == add
   interpolate(xlist, ylist, m, k) ==

First we check whether we have the right number of points and values. Clearly the number
of points and the number of values must be identical. Note that we want to determine the
numerator and denominator polynomials only up to a factor. Thus, we want to determine
In fact, we could also leave $k$ unspecified and determine it as $k = \#\text{list} - m - 1$:

I don’t know whether this would be better.

The next step is to set up the matrix. Suppose that our numerator polynomial is $p(x) = a_0 + a_1 x + \ldots + a_m x^m$ and that our denominator polynomial is $q(x) = b_0 + b_1 x + \ldots + b_m x^m$.

Then we have the following equations, writing $n$ for $m + k + 1$:

\[
\begin{align*}
    p(x_1) - y_1 q(x_1) &= a_0 + a_1 x_1 + \ldots + a_m x_1^m - y_1 (b_0 + b_1 x_1 + \ldots + b_k x_1^k) = 0 \\
    p(x_2) - y_2 q(x_2) &= a_0 + a_1 x_2 + \ldots + a_m x_2^m - y_2 (b_0 + b_1 x_2 + \ldots + b_k x_2^k) = 0 \\
    \vdots \\
    p(x_n) - y_n q(x_n) &= a_0 + a_1 x_n + \ldots + a_m x_n^m - y_n (b_0 + b_1 x_n + \ldots + b_k x_n^k) = 0
\end{align*}
\]

This can be written as

\[
\begin{bmatrix}
    1 & x_1 & \cdots & x_1^m & -y_1 & -y_1 x_1 & \cdots & -y_1 x_1^k \\
    1 & x_2 & \cdots & x_2^m & -y_2 & -y_2 x_2 & \cdots & -y_2 x_2^k \\
    \vdots \\
    1 & x_n & \cdots & x_n^m & -y_n & -y_n x_n & \cdots & -y_n x_2^k
\end{bmatrix}
\begin{bmatrix}
    a_0 \\
    a_1 \\
    \vdots \\
    a_m \\
    b_0 \\
    b_1 \\
    \vdots \\
    b_k
\end{bmatrix} = 0
\]

We generate this matrix columnwise:
Now we can solve the system:
   — package RINTERP RationalInterpolation —
   
   res: List Vector F := nullSpace((transpose matrix collist) _
   ::Matrix F)
   
   Note that it may happen that the system has several solutions. In this case, some of the
   data points may not be interpolated correctly. However, the solution is often still useful,
   thus we do not signal an error.
   
   — package RINTERP RationalInterpolation —
   
   if #res = 1 then output("Warning: unattainable points!" _
   ::OutputForm)$OutputPackage
   
   In this situation, all the solutions will be equivalent, thus we can always simply take the first
   one:
   
   — package RINTERP RationalInterpolation —
   
   reslist: List List Polynomial F := _
   [[(res.1).(i+1)*(xx::Polynomial F)**i for i in 0..m], _
   [(res.1).(i+m+2)*(xx::Polynomial F)**i for i in 0..k]]
   
   Finally, we generate the rational function:
   
   — package RINTERP RationalInterpolation —
   
   reduce(_,reslist.1)/reduce(_,reslist.2)

"RINTERP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RINTERP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RINTERP" -> "PFECAT"
package ODERAT RationalLODE

--- RationalLODE.input ---

)set break resume
)sys rm -f RationalLODE.output
)spool RationalLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalLODE
--E 1

)spool
)lisp (bye)

---

--- RationalLODE.help ---

====================================================================
RationalLODE examples
====================================================================

RationalLODE provides functions for in-field solutions of linear ordinary differential equations, in the rational case.

See Also:
  o )show RationalLODE

---
RationalLODE (ODERAT)

Exports:
indicialEquationAtInfinity  ratDsolve

— package ODERAT RationalLODE —

)abbrev package ODERAT RationalLODE
++ Author: Manuel Bronstein
++ Date Created: 13 March 1991
++ Date Last Updated: 13 April 1994
++ Description:
++ \spad{RationalLODE} provides functions for in-field solutions of linear
++ ordinary differential equations, in the rational case.

RationalLODE(F, UP): Exports == Implementation where
F : Join(Field, CharacteristicZero, RetractableTo Integer,
    RetractableTo Fraction Integer)
UP : UnivariatePolynomialCategory F
N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
U ==> Union(RF, "failed")
V ==> Vector F
M ==> Matrix F
LODO ==> LinearOrdinaryDifferentialOperator1 RF
LODO2==> LinearOrdinaryDifferentialOperator2(UP, RF)

Exports == with
ratDsolve: (LODO, RF) -> Record(particular: U, basis: List RF)
  ++ ratDsolve(op, g) returns \spad{{"failed", []}} if the equation
  ++ \spad{op y = g} has no rational solution. Otherwise, it returns
  ++ \spad{[f, [yi, \ldots, ym]]} where f is a particular rational solution
  ++ and the yi’s form a basis for the rational solutions of the
  ++ homogeneous equation.
ratDsolve: (LODO, List RF) -> Record(basis:List RF, mat:Matrix F)
++ ratDsolve(op, [g1,...,gm]) returns \spad{[[h1,...,hq], M]} such ++ that any rational solution of \spad{op y = c1 g1 + ... + cm gm} ++ is of the form \spad{d1 h1 + ... + dq hq} where ++ \spad{M [d1,...,dq,c1,...,cm] = 0}.
ratDsolve: (LODO2, RF) -> Record(particular: U, basis: List RF) ++ ratDsolve(op, g) returns \spad{["failed", []]} if the equation ++ \spad{op y = g} has no rational solution. Otherwise, it returns ++ \spad{[f, [y1,...,ym]]} where f is a particular rational solution ++ and the yi’s form a basis for the rational solutions of the ++ homogeneous equation.
ratDsolve: (LODO2, List RF) -> Record(basis:List RF, mat:Matrix F) ++ ratDsolve(op, [g1,...,gm]) returns \spad{[[h1,...,hq], M]} such ++ that any rational solution of \spad{op y = c1 g1 + ... + cm gm} ++ is of the form \spad{d1 h1 + ... + dq hq} where ++ \spad{M [d1,...,dq,c1,...,cm] = 0}.

Implementation ==> add
import BoundIntegerRoots(F, UP)
import RationalIntegration(F, UP)
import PrimitiveRatDE(F, UP, LODO2, LODO)
import LinearSystemMatrixPackage(F, V, V, M)
import InnerCommonDenominator(UP, RF, List UP, List RF)

infOrder f == (degree denom f) - (degree numer f)
evenodd n == (even? n => 1; -1)
ratDsolve1(op, lg) ==
\[
\begin{align*}
d &:= \text{denomLODE}(\text{op}, \lg) \\
\text{rec} &:= \text{candidates}(\text{op}, \lg, d) \\
l &:= \text{concat}([\text{op} \ q \text{ for } q \text{ in rec.basis}], \\
&\quad [\text{op}(\text{rec.particular}.i) - \lg.i \text{ for } i \text{ in } 1..\#(\text{rec.particular})]) \\
\text{sys1} &:= \text{reducedSystem}([l])\@\text{Matrix}(\text{UP}) \\
[\text{rec.basis}, \text{reducedSystem sys1}] \\
\text{ratDsolve0}(\text{op}, g) &= \text{zero? degree op} 
\Rightarrow [\text{inv(leadingCoefficient(op)::RF)} * g, \text{empty()}] \\
\text{minimumDegree op} > 0 &\Rightarrow \\
\text{sol} &:= \text{ratDsolve0}(\text{monicRightDivide}([\text{op}, \text{monomial}(1, 1)].\text{quotient}, g) \\
b:\text{List(RF)} := [1] \\
\text{for } f \text{ in } \text{sol.basis} \text{ repeat} \\
\text{if } (\text{uu} := \text{inffieldint f}) \text{ case RF then } b := \text{concat}(\text{uu::RF}, b) \\
\text{sol.particular} \text{ case "failed" } \Rightarrow ["\text{failed"}, b] \\
[\text{inffieldint(sol.particular::RF)}, b] \\
(\text{u := denomLODE}(\text{op}, g)) \text{ case "failed" } \Rightarrow ["\text{failed"}, \text{empty()}] \\
\text{rec} &:= \text{candidates}(\text{op}, [g], \text{u::UP}) \\
l &:= 1 \text{b} := \text{lsol} := \text{empty()}$\text{List(RF)} \\
\text{for } q \text{ in } \text{rec.basis} \text{ repeat} \\
\text{if } \text{zero?(opq := op q) then } \text{lsol} := \text{concat}(q, \text{lsol}) \\
\text{else } (l := \text{concat}(\text{opq}, 1); lb := \text{concat}(q, lb)) \\
h:RF &:= (\text{zero? g} \Rightarrow 0; \text{first(\text{rec.particular})}) \\
\text{empty? l} &\Rightarrow \\
\text{zero? g} &\Rightarrow [0, \text{lsol}] \\
&[(g = \text{op h} \Rightarrow h; "\text{failed"}), \text{lsol}] \\
m:\text{M} \\
v:V \\
\text{if } \text{zero? g} \text{ then} \\
\text{m} &:= \text{reducedSystem}(\text{reducedSystem}([l])\@\text{Matrix}(\text{UP}))\@\text{M} \\
v &:= \text{new}(\text{ncols m}, 0)$V \\
\text{else} \\
\text{sys1} &:= \text{reducedSystem}([l], \text{vector } [g - \text{op h}]) \\
\text{sys2} &:= \text{reducedSystem}([\text{sys1.mat}, \text{sys1.vec}]\@\text{Record(mat: Matrix UP, vec: Vector UP)}) \\
\text{m} &:= \text{sys2.mat} \\
v &:= \text{sys2.vec} \\
\text{sol} &:= \text{solve}(\text{m}, v) \\
\text{part:U} &:= \\
\text{zero? g} &\Rightarrow 0 \\
\text{sol.particular} \text{ case "failed" } \Rightarrow "\text{failed"} \\
\text{makeDot}(\text{sol.particular::V}, lb) + \text{first(\text{rec.particular})} \\
[\text{part}, \\
\text{concat}_{-1}([\text{lsol}, \text{makeDot}(v, lb) \text{ for } \text{v in sol.basis | nzero? v}])] \\
\text{indicialEquationAtInfinity}(\text{op}:\text{LODO2}) &= \\
\text{rec} :&= \text{infMULambda op} \\
\text{infIndicialEquation}(\text{rec.lambda}, \text{rec.func}) \\
\text{indicialEquationAtInfinity}(\text{op}:\text{LODO}) &= 
\end{align*}
\]
rec := splitDenominator(op, empty())
indicialEquationAtInfinity(rec.eq)

regularPoint(l, lg) ==
a := leadingCoefficient(l) * commonDenominator lg
coefficient(a, 0) ^= 0 => 0
for i in 1.. repeat
  a(j := i::F) ^= 0 => return i
  a(-j) ^= 0 => return(-i)

unitlist(i, q) ==
v := new(q, 0)$Vector(F)
v.i := 1
parts v

candidates(op, lg, d) ==
n := degree d + infBound(op, lg)
m := regularPoint(op, lg)
uts := UnivariateTaylorSeries(F, dummy, m::F)
tools := UTSodeTools(F, UP, LODO2, uts)
solver := UnivariateTaylorSeriesODESolver(F, uts)
dd := UP2UTS(d)$tools
f := LODO2FUN(op)$tools
q := degree op
e := unitlist(1, q)
hom := [UTS2UP(dd * ode(f, unitlist(i, q))$solver, n)$tools /$RF d
  for i in 1..q]$List(RF)
a1 := inv(leadingCoefficient(op)::RF)
part :=
  [UTS2UP(dd *
    ode((l1:List(uts)):uts +->
      RF2UTS(a1 * g)$tools + f l1, e)$solver, n)$tools
    /$RF d for g in lg | g ^= 0]$List(RF)

nzero? v ==
  for i in minIndex v .. maxIndex v repeat
    not zero? qelt(v, i) => return true
  false

-- returns z(z+1)...(z+(n-1))
UPfact n ==
  zero? n => 1
  z := monomial(1, 1)$UP
  *[Z + i::F::UP for i in 0..(n-1)::N]

infMuLambda l ==
lamb:List(N) := [d := degree l]
lf:List(UP) := [a := leadingCoefficient l]
mup := degree(a)::Z - d
while (l := reductum l) ^= 0 repeat
  a := leadingCoefficient l
  if (m := degree(a)::Z - (d := degree l)) > mup then
    mup := m
    lamb := [d]
    lf := [a]
  else if (m = mup) then
    lamb := concat(d, lamb)
    lf := concat(a, lf)
  [mup, lamb, lf]

infIndicialEquation(lambda, lf) ==
ans:UP := 0
for i in lambda for f in lf repeat
  ans := ans + evenodd i * leadingCoefficient f * UPfact i
ans

infBound(l, lg) ==
rec := infMuLambda l
n := min(- degree(l)::Z - 1,
  integerBound infIndicialEquation(rec.lambda, rec.func))
while not(empty? lg) and zero? first lg repeat lg := rest lg
empty? lg => (-n)::N
m := infOrder first lg
for g in rest lg repeat
  if not(zero? g) and (mm := infOrder g) < m then m := mm
(-min(n, rec.mu - degree(leadingCoefficient l)::Z + m))::N

makeDot(v, bas) ==
ans:RF := 0
for i in 1.. for b in bas repeat ans := ans + v.i::UP * b
ans

ratDsolve(op:LODO, g:RF) ==
rec := splitDenominator(op, [g])
ratDsolve0(rec.eq, first(rec.rh))
ratDsolve(op:LODO2, g:RF) ==
unit?(c := content op) => ratDsolve0(op, g)
ratDsolve0((op exquo c)::LODO2, inv(c::RF) * g)
ratDsolve(op:LODO2, lg:List RF) ==
unit?(c := content op) => ratDsolve1(op, lg)
ratDsolve1((op exquo c)::LODO2, [inv(c::RF) * g for g in lg])
package RATRET RationalRetractions

---

RationalRetractions.examples

---

Rational number testing and retraction functions.

See Also:
- )show RationalRetractions
RationalRetractions (RATRET)

Exports:
rational rational? rationalIfCan

— package RATRET RationalRetractions —

)abbrev package RATRET RationalRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Rational number testing and retraction functions.

RationalRetractions(S:RetractableTo(Fraction Integer)): with
    rational : S -> Fraction Integer
    ++ rational(x) returns x as a rational number;
    ++ error if x is not a rational number;
    rational? : S -> Boolean
    ++ rational?(x) returns true if x is a rational number,
    ++ false otherwise;
    rationalIfCan: S -> Union(Fraction Integer, "failed")
    ++ rationalIfCan(x) returns x as a rational number,
    ++ "failed" if x is not a rational number;
== add
    rational s == retract s
    rational? s == retractIfCan(s) case Fraction(Integer)
    rationalIfCan s == retractIfCan s

— RATRET.dotabb —

"RATRET" [color=FF4488,href=bookvol10.4.pdf#nameddest=RATRET]
"PID" [color=4488FF,href=bookvol10.2.pdf#nameddest=PID]
package ODERTRIC RationalRicDE

— RationalRicDE.input —

)set break resume
)sys rm -f RationalRicDE.output
)spool RationalRicDE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalRicDE
--E 1

)spool
)lisp (bye)

— RationalRicDE.help —

====================================================================
RationalRicDE examples
====================================================================

In-field solution of Riccati equations, rational case.

See Also:
o )show RationalRicDE

———
RationalRicDE (ODERTRIC)

Exports:
polyRicDE  ricDsolve  singRicDE

— package ODERTRIC RationalRicDE —

)abbrev package ODERTRIC RationalRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ In-field solution of Riccati equations, rational case.

RationalRicDE(F, UP): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer,
            RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

N     ==> NonNegativeInteger
Z     ==> Integer
SY    ==> Symbol
P     ==> Polynomial F
RF    ==> Fraction P
EQ    ==> Equation RF
QF    ==> Fraction UP
UP2   ==> SparseUnivariatePolynomial UP
SUP   ==> SparseUnivariatePolynomial P
REC   ==> Record(poly:SUP, vars:List SY)
SOL   ==> Record(var:List SY, val:List F)
POL   ==> Record(poly:UP, eq:L)
FRC   ==> Record(frac:QF, eq:L)
CNT   ==> Record(constant:F, eq:L)
UTS   ==> UnivariateTaylorSeries(F, dummy, 0)
UPS   ==> SparseUnivariatePolynomial UTS
L     ==> LinearOrdinaryDifferentialOperator2(UP, QF)
LQ    ==> LinearOrdinaryDifferentialOperator1 QF
Exports ==>

with

ricDsolve: (LQ, UP -> List F) -> List QF
++ ricDsolve(op, zeros) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).
++ \( \text{spad}(\text{zeros}) \) is a zero finder in \( \text{spad}(\text{UP}) \).

ricDsolve: (LQ, UP -> List F, UP -> Factored UP) -> List QF
++ ricDsolve(op, zeros, ezfactor) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).
++ \( \text{spad}(\text{zeros}) \) is a zero finder in \( \text{spad}(\text{UP}) \).
++ Argument \( \text{spad}(\text{ezfactor}) \) is a factorisation in \( \text{spad}(\text{UP}) \), not necessarily into irreducibles.

ricDsolve: (L, UP -> List F) -> List QF
++ ricDsolve(op, zeros) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).
++ \( \text{spad}(\text{zeros}) \) is a zero finder in \( \text{spad}(\text{UP}) \).

ricDsolve: (L, UP -> List F, UP -> Factored UP) -> List QF
++ ricDsolve(op, zeros, ezfactor) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).
++ \( \text{spad}(\text{zeros}) \) is a zero finder in \( \text{spad}(\text{UP}) \).
++ Argument \( \text{spad}(\text{ezfactor}) \) is a factorisation in \( \text{spad}(\text{UP}) \), not necessarily into irreducibles.

singRicDE: (L, UP -> Factored UP) -> List FRC
++ singRicDE(op, ezfactor) returns \( \text{spad}([[f1,L1], [f2,L2], ..., [fk,Lk]]) \)
++ such that the singular ++ part of any rational solution of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \) must be one of the fi’s (up to the constant coefficient), in which case the equation for \( \text{spad}(z = y e^{-\text{int ai}}) \) is \( \text{spad}(\text{Li z = 0}) \).
++ Argument \( \text{spad}(\text{ezfactor}) \) is a factorisation in \( \text{spad}(\text{UP}) \), not necessarily into irreducibles.

polyRicDE: (L, UP -> List F) -> List POL
++ polyRicDE(op, zeros) returns \( \text{spad}([[p1,L1], [p2,L2], ..., [pk,Lk]]) \)
++ such that the polynomial part of any rational solution of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \) must be one of the pi’s (up to the constant coefficient), in which case the equation for \( \text{spad}(z = y e^{-\text{int p}}) \) is \( \text{spad}(\text{Li z = 0}) \).
++ \( \text{spad}(\text{zeros}) \) is a zero finder in \( \text{spad}(\text{UP}) \).

if \( F \) has AlgebraicallyClosedField then

ricDsolve: LQ -> List QF
++ ricDsolve(op) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).

ricDsolve: (LQ, UP -> Factored UP) -> List QF
++ ricDsolve(op, ezfactor) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).
++ Argument \( \text{spad}(\text{ezfactor}) \) is a factorisation in \( \text{spad}(\text{UP}) \), not necessarily into irreducibles.

ricDsolve: L -> List QF
++ ricDsolve(op) returns the rational solutions of the associated Riccati equation of \( \text{spad}(\text{op y = 0}) \).

ricDsolve: (L, UP -> Factored UP) -> List QF
++ `ricDsolve(op, ezfactor)` returns the rational solutions of the
++ associated Riccati equation of \( \text{spad}{op \ y = 0} \).
++ Argument \( \text{spad}{ezfactor} \) is a factorisation in \( \text{spad}{UP} \),
++ not necessarily into irreducibles.

Implementation ==> add
import RatODETools(P, SUP)
import RationalLODE(F, UP)
import NonLinearSolvePackage F
import PrimitiveRatDE(F, UP, L, LQ)
import PrimitiveRatRicDE(F, UP, L, LQ)
FifCan : RF -> Union(F, "failed")
UP2SUP : UP -> SUP
innersol : (List UP, Boolean) -> List QF
mapeval : (SUP, List SY, List F) -> UP
ratsol : List List EQ -> List SOL
ratsln : List EQ -> Union(SOL, "failed")
solveModulo : (UP, UP2) -> List UP
logDerOnly : L -> List QF
nonSingSolve : (N, L, UP -> List F) -> List QF
constantRic : (UP, UP -> List F) -> List F
nopoly : (N, UP, L, UP -> List F) -> List QF
reverseUP : UP -> UTS
reverseUTS : (UTS, N) -> UP
newtonSolution : (L, F, N, UP -> List F) -> UP
newtonSolve : (UPS, F, N) -> Union(UTS, "failed")
genericPolynomial: (SY, Z) -> Record(poly:SUP, vars:List SY)
-- genericPolynomial(s, n) returns
-- \( \text{spad}[[[s0 + s1 X +...+ sn X^n],[s0,...,sn]]} \).
dummy := new()$SY

UP2SUP p == map(z +-> z::P,p)
$logDerOnly l == [differentiate(s) / s for s in ratDsolve(l, 0).basis]
ricDsolve(l:LQ, zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)
ricDsolve(l:L, zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)
singRicDE(l, ezfactor) == singRicDE(l, ezfactor)
ricDsolve(l:LQ, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
ricDsolve(splitDenominator(l, empty()).eq, zeros, ezfactor)
mapeval(p, ls, lv) ==
map(z +-> ground eval(z, ls, lv),p)
$UnivariatePolynomialCategoryFunctions2(P, SUP, F, UP)
FifCan f ==
((n := retractIfCan(numer f))@Union(F, "failed") case F) and
((d := retractIfCan(denom f))@Union(F, "failed") case F) =>
\[(n::F) / (d::F) \]
"failed"

-- returns [0, []] if n < 0

\[\text{genericPolynomial}(s, n) ==\]
\[\text{ans: SUP := 0}\]
\[\text{l: List(SY) := empty()}\]
\[\text{for i in 0..n repeat}\]
\[\text{ans := ans + monomial((sy := new s)::P, i::N)}\]
\[\text{l := concat(sy, l)}\]
\[\text{[ans, reverse_! l]}\]

\[\text{ratsln l ==}\]
\[\text{ls: List(SY) := empty()}\]
\[\text{lv: List(F) := empty()}\]
\[\text{for eq in l repeat}\]
\[\text{((u := FfCan rhs eq) case "failed") or}\]
\[\text{((v := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed")}\]
\[\Rightarrow \text{return "failed"}\]
\[\text{lv := concat(u::F, lv)}\]
\[\text{ls := concat(v::SY, ls)}\]
\[\text{[ls, lv]}\]

\[\text{ratsol l ==}\]
\[\text{ans: List(SOL) := empty()}\]
\[\text{for sol in l repeat}\]
\[\text{if ((u := ratsln sol) case SOL) then ans := concat(u::SOL, ans)}\]
\[\text{ans}\]

-- returns [] if the solutions of l have no polynomial component

\[\text{polyRicDE(l, zeros) ==}\]
\[\text{ans: List(POL) := [[0, l]]}\]
\[\text{empty?(lc := leadingCoefficientRicDE l) => ans}\]
\[\text{rec := first lc} -- one with highest degree\]
\[\text{for a in zeros(rec.eq) | a ^= 0 repeat}\]
\[\text{if (p := newtonSolution(l, a, rec.deg, zeros)) ^= 0 then}\]
\[\text{ans := concat([p, changeVar(l, p)], ans)}\]
\[\text{ans}\]

-- reverseUP(a_0 + a_1 x + ... + an x^n) = a_n + ... + a_0 x^n

\[\text{reverseUP p ==}\]
\[\text{ans: UTS := 0}\]
\[\text{n := degree(p)::Z}\]
\[\text{while p ^= 0 repeat}\]
\[\text{ans := ans + monomial(leadingCoefficient p, (n - degree p)::N)}\]
\[\text{p := reductum p}\]
\[\text{ans}\]

-- reverseUTS(a_0 + a_1 x + ..., n) = a_n + ... + a_0 x^n

\[\text{reverseUTS(s, n) ==}\]
newtonSolution(l, a, n, zeros) ==
  i:N
  m:Z := 0
  aeq:UPS := 0
  op := l
  while op ^= 0 repeat
    mu := degree(op) * n + degree leadingCoefficient op
    op := reductum op
    if mu > m then m := mu
  while l ^= 0 repeat
    c := leadingCoefficient l
    d := degree l
    s:UTS := monomial(1, (m - d * n - degree c)::N)$UTS * reverseUP c
    aeq := aeq + monomial(s, d)
    l := reductum l
  (u := newtonSolve(aeq, a, n)) case UTS => reverseUTS(u::UTS, n)
-- newton lifting failed, so revert to traditional method
  atn := monomial(a, n)$UP
  neq := changeVar(l, atn)
  sols := [sol.poly for sol in polyRicDE(neq, zeros) | degree(sol.poly) < n]
  empty? sols => atn
  atn + first sols

newtonSolve(eq, a, n) ==
  deq := differentiate eq
  sol := a::UTS
  for i in 1..n repeat
    (xquo := eq(sol) exquo deq(sol)) case "failed" => return "failed"
    sol := truncate(sol - xquo::UTS, i)
  sol

ricDsolve(l:L, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
  n := degree l
  ans:List(QF) := empty()
  for rec in singRicDE(l, zeros) repeat
    ans := removeDuplicates_!(concat_!(ans,
      [rec.frac + f for f in nonSingSolve(n, rec.eq, zeros)])
    )
  #ans = n => return ans
  ans
-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
nonSingSolve(n, l, zeros) ==
    ans:List(QF) := empty()
    for rec in polyRicDE(l, zeros) repeat
        ans := removeDuplicates!(concat!(ans, nopoly(n, rec.poly, rec.eq, zeros)))
    if #ans = n => return ans
    ans

constantRic(p, zeros) ==
    zero? degree p => empty()
    zeros squareFreePart p

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
nopoly(n, p, l, zeros) ==
    ans:List(QF) := empty()
    for rec in constantCoefficientRicDE(l, z +-> constantRic(z, zeros)) repeat
        ans := removeDuplicates!(concat!(ans, [(rec.constant::UP + p)::QF + f for f in logDerOnly(rec.eq)]))
    if #ans = n => return ans
    ans

-- returns [p1,...,pn] s.t. h(x,pi(x)) = 0 mod c(x)
solveModulo(c, h) ==
    rec := genericPolynomial(dummy, degree(c)::Z - 1)
    unk: SUP := 0
    while not zero? h repeat
        unk := unk + UP2SUP(leadingCoefficient h) * (rec.poly ** degree h)
        h := reductum h
    sol := ratsol solve(coefficients(monicDivide(unk, UP2SUP c).remainder),
                       rec.vars)
    [map(eval(rec.poly, s.var, s.val) for s in sol)
     if F has AlgebraicallyClosedField then
         zro1: UP -> List F
         zro : (UP, UP -> Factored UP) -> List F
         ricDsolve(l:L) == ricDsolve(l, squareFree)
         ricDsolve(l:LQ) == ricDsolve(l, squareFree)
         ricDsolve(l:L, ezfactor:UP -> Factored UP) ==
             ricDsolve(l, z +-> zro(z, ezfactor), ezfactor)
         ricDsolve(l:LQ, ezfactor:UP -> Factored UP) ==
             ricDsolve(l, z +-> zro(z, ezfactor), ezfactor)
         zro(p, ezfactor) ==
             concat [zro1(r.factor) for r in factors ezfactor p]
zro1 p ==
[zeroOf(map((z:F):F +-> z, p)
  $\text{UnivariatePolynomialCategoryFunctions2}(F, UP, F,
  \text{SparseUnivariatePolynomial} F))]

---

--- ODERTRIC.dotabb ---

"ODERTRIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODERTRIC"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"ODERTRIC" -> "UTSCAT"

---

package RURPK RationalUnivariateRepresentationPackage

--- RationalUnivariateRepresentationPackage.input ---

)set break resume
)sys rm -f RationalUnivariateRepresentationPackage.output
)spool RationalUnivariateRepresentationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalUnivariateRepresentationPackage

--E 1

)spool
)lisp (bye)

---

--- RationalUnivariateRepresentationPackage.help ---

====================================================================
RationalUnivariateRepresentationPackage examples
====================================================================

A package for computing the rational univariate representation
of a zero-dimensional algebraic variety given by a regular
triangular set. This package is essentially an interface for the
InternalRationalUnivariateRepresentationPackage constructor.
It is used in the ZeroDimensionalSolvePackage for solving polynomial
systems with finitely many solutions.

See Also:
o )show RationalUnivariateRepresentationPackage

RationalUnivariateRepresentationPackage (RURPK)

RURPK

SFRTCAT

Exports:
rur

--- package RURPK RationalUnivariateRepresentationPackage ---

)abbrev package RURPK RationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ Description:
++ A package for computing the rational univariate representation
++ of a zero-dimensional algebraic variety given by a regular
++ triangular set. This package is essentially an interface for the
++ \spadtype{InternalRationalUnivariateRepresentationPackage} constructor.
++ It is used in the \spadtype{ZeroDimensionalSolvePackage}
++ for solving polynomial systems with finitely many solutions.

RationalUnivariateRepresentationPackage(R,ls): Exports == Implementation where
R : Join(EuclideanDomain,CharacteristicZero)
ls: List Symbol


```
N ==> NonNegativeInteger
Z ==> Integer
P ==> Polynomial R
LP ==> List P
U ==> SparseUnivariatePolynomial(R)
RUR ==> Record(complexRoots: U, coordinates: LP)

Exports == with

rur: (LP,Boolean) -> List RUR
++ \spad{rur(lp,univ?)} returns a rational univariate representation
++ of \spad{lp}. This assumes that \spad{lp} defines a regular
++ triangular \spad{ts} whose associated variety is zero-dimensional
++ over \spad{R}. \spad{rur(lp,univ?)} returns a list of items
++ \spad{\{u,lc\}} where \spad{u} is an irreducible univariate polynomial
++ and each \spad{c} in \spad{lc} involves two variables: one from \spad{ls},
++ called the coordinate of \spad{c}, and an extra variable which
++ represents any root of \spad{u}. Every root of \spad{u} leads to
++ a tuple of values for the coordinates of \spad{lc}. Moreover,
++ a point \spad{x} belongs to the variety associated with \spad{lp} iff
++ there exists an item \spad{\{u,lc\}} in \spad{rur(lp,univ?)} and
++ a root \spad{r} of \spad{u} such that \spad{x} is given by the
++ tuple of values for the coordinates of \spad{lc} evaluated at \spad{r}.
++ If \spad{univ?} is \spad{true} then each polynomial \spad{c}
++ will have a constant leading coefficient w.r.t. its coordinate.
++ See the example which illustrates the \spad{ZeroDimensionalSolvePackage}
++ package constructor.

rur: (LP) -> List RUR
++ \spad{rur(lp)} returns the same as \spad{rur(lp,true)}

rur: (LP,Boolean,Boolean) -> List RUR
++ \spad{rur(lp,univ?,check?)} returns the same as \spad{rur(lp,true)}.
++ Moreover, if \spad{check?} is \spad{true} then the result is checked.
```

Implementation == add

```
news: Symbol := new()$Symbol
lv: List Symbol := concat(ls,news)
V ==> OrderedVariableList(lv)
Q ==> NewSparseMultivariatePolynomial(R,V)
E ==> IndexedExponents V
TS ==> SquareFreeRegularTriangularSet(R,E,V,Q)
QWT ==> Record(val: Q, tower: TS)
LQWT ==> Record(val: List Q, tower: TS)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,Q)
normpack ==> NormalizationPackage(R,E,V,Q,TS)
rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E,V,Q,TS)
newv: V := variable(news)::V
newq : Q := newv :: Q

rur(lp: List P, univ?: Boolean, check?: Boolean): List RUR ==
lp := remove(zero?,lp)
```

empty? lp =>
    error "rur$RURPACK: #1 is empty"
any?(ground?,lp) =>
    error "rur$RURPACK: #1 is not a triangular set"

ts: TS := [[newq]$(List Q)]
lq: List Q := []
for p in lp repeat
    rif: Union(Q,"failed") := retractIfCan(p)$Q
    rif case "failed" =>
        error "rur$RURPACK: #1 is not a subset of R[ls]"
    q: Q := rif::Q
    lq := cons(q,lq)
lq := sort(infRittWu?,lq)
toSee: List LQWT := [[lq,ts]$LQWT]
toSave: List TS := []
while not empty? toSee repeat
    lqwt := first toSee; toSee := rest toSee
    lq := lqwt.val; ts := lqwt.tower
    empty? lq =>
        -- output(ts::OutputForm)$OutputPackage
        toSave := cons(ts,toSave)
    q := first lq; lq := rest lq
    not (mvar(q) > mvar(ts)) =>
        error "rur$RURPACK: #1 is not a triangular set"
    empty? (rest(ts)::TS) =>
        lfq := irreducibleFactors([q])$polsetpack
        for fq in lfq repeat
            newts := internalAugment(fq,ts)
            newlq := [remainder(q,newts).polnum for q in lq]
            toSee := cons([newlq,newts]$LQWT,toSee)
    lsfqwt: List QWT := squareFreePart(q,ts)
    for qwt in lsfqwt repeat
        q := qwt.val; ts := qwt.tower
        if not ground? init(q)
            then
                q := normalizedAssociate(q,ts)$normpack
                newts := internalAugment(q,ts)
                newlq := [remainder(q,newts).polnum for q in lq]
                toSee := cons([newlq,newts]$LQWT,toSee)
    toReturn: List RUR := []
    for ts in toSave repeat
        lus := rur(ts,univ?)$rurpack
        check? and (not checkRur(ts,lus)$rurpack) =>
            output("RUR for: ")$OutputPackage
            output(ts::OutputForm)$OutputPackage
            output("Is: ")$OutputPackage
            for us in lus repeat output(us::OutputForm)$OutputPackage
            error "rur$RURPACK: bad result with function rur$IRURPK"
    for us in lus repeat
        g: U := univariate(select(us,newv)::Q)$Q
package POLUTIL RealPolynomialUtilitiesPackage

This file describes the Real Closure 1.0 package which consists of different packages, categories and domains.

The package RealPolynomialUtilitiesPackage which receives a field and a univariate polynomial domain with coefficients in the field. It computes some simple functions such as Strum and Sylvester sequences.

The category RealRootCharacterizationCategory provides abstract functionalities to work with "real roots" of univariate polynomials. These resemble variables with some functionalities needed to compute important operations.

RealClosedField is a category which provides common operations available over real closed fields. These include finding all the roots of univariate polynomial, taking square roots, ...

CAVEATS
Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like sqrt(2) + sqrt(2) you have two new variables which happen to be equal. To avoid this name the expression such as in s2 := sqrt(2) : s2 + s2

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which most natural to you.

LIMITATIONS
The file reclos.input show some basic use of the package. This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Beta versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several “real roots”.

--- RealPolynomialUtilitiesPackage.input ---

)set break resume
)sys rm -f RealPolynomialUtilitiesPackage.output
)spool RealPolynomialUtilitiesPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RealPolynomialUtilitiesPackage
--E 1

)spool
)lisp (bye)

---

--- RealPolynomialUtilitiesPackage.help ---

====================================================================
RealPolynomialUtilitiesPackage examples
====================================================================

RealPolynomialUtilitiesPackage provides common functions used by interval coding.

See Also:
  o )show RealPolynomialUtilitiesPackage

---
RealPolynomialUtilitiesPackage (POLUTIL)

Exports:
  boundOfCauchy  lazyVariations  sturmSequence  sturmVariationsOf
  sylvesterSequence

— package POLUTIL RealPolynomialUtilitiesPackage —

)abbrev package POLUTIL RealPolynomialUtilitiesPackage
++ Author: Renaud Rioboo
++ Date Created: summer 1992
++ Description:
++ \axiomType{RealPolynomialUtilitiesPackage} provides common functions used
++ by interval coding.

RealPolynomialUtilitiesPackage(TheField,ThePols) : PUB == PRIV where

  TheField : Field
  ThePols : UnivariatePolynomialCategory(TheField)

  Z ==> Integer
  N ==> NonNegativeInteger
  P ==> ThePols

  PUB == with

    sylvesterSequence : (ThePols,ThePols) -> List ThePols
      ++ \axiom{sylvesterSequence(p,q)} is the negated remainder sequence
      ++ of p and q divided by the last computed term
    sturmSequence : ThePols -> List ThePols
      ++ \axiom{sturmSequence(p) = sylvesterSequence(p,p')}
    if TheField has OrderedRing then
      boundOfCauchy : ThePols -> TheField
        ++ \axiom{boundOfCauchy(p)} bounds the roots of p
      sturmVariationsOf : List TheField -> N
        ++ \axiom{sturmVariationsOf(l)} is the number of sign variations
        ++ in the list of numbers l,
++ note that the first term counts as a sign
lazyVariations : (List(TheField), Z, Z) -> N
++ \( \text{axiom}\{\text{lazyVariations}(l,s1,sn)\}\) is the number of sign variations
++ in the list of non null numbers \([s1::l]@sn\),

PRIV == add

sturmSequence(p) ==
  sylvesterSequence(p,differentiate(p))

sylvesterSequence(p1,p2) ==
  res : List(ThePols) := \[p1\]
  while \((p2 ^= 0)\) repeat
    res := cons(p2 , res)
    (p1 , p2) := (p2 , -(p1 rem p2))
  if degree(p1) > 0
    then
      p1 := unitCanonical(p1)
      res := \[ term quo p1 for term in res \]
      reverse! res
  if TheField has OrderedRing
  then
    boundOfCauchy(p) ==
      c :TheField := inv(leadingCoefficient(p))
      l := \[ c*term for term in rest(coefficients(p))\]
      null(l) => 1
      1 + ("max" / \[ abs(t) for t in l \])

-- sturmVariationsOf(l) ==
-- res : N := 0
-- lsg := sign(first(l))
-- for term in l repeat
--   if "( (sg := sign(term) ) = 0 ) then
--     if (sg ^= lsg) then res := res + 1
--     lsg := sg
--   res

sturmVariationsOf(l) ==
  null(l) => error "POLUTIL: sturmVariationsOf: empty list !"
  l1 := first(l)
  -- first 0 counts as a sign
  l1 : List(TheField) := []
  for term in rest(l1) repeat
    -- zeros don't count
    if not(zero?(term)) then l1 := cons(term,l1)
  if l1 is not zero then l1 = reverse(l1)
  null(l1) => error "POLUTIL: sturmVariationsOf: Bad sequence"
\begin{verbatim}
ln := first(ll)
ll := reverse(rest(ll))
-- if l1 is not zero then first(l) = first(ll)
-- if l1 is zero then first zero should count as a sign
zero?(l1) => 1 + lazyVariations(rest(ll),sign(first(ll)),sign(ln))
lazyVariations(ll, sign(l1), sign(ln))

lazyVariations(l,s1,s2) ==
zero?(s1) or zero?(s2) => error "POLUTIL: lazyVariations: zero sign!"
null(l) =>
  if s1 = s2 then 0 else 1
null(rest(l)) =>
  if zero?(first(l))
  then error "POLUTIL: lazyVariations: zero sign!"
  else
    if s1 = s2
      then if (s1 = sign(first(l)))
          then 0
            else 2
          -- in this case we save one test
        else 1
  s := sign(l.2)
lazyVariations([first(l)],s1,s) +
lazyVariations(rest(rest(l)),s,s2)
\end{verbatim}

---

\section{POLUTIL.dotabb}

"POLUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLUTIL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLUTIL" -> "PFECAT"

---

\section{package REALSOLV RealSolvePackage}

\begin{verbatim}
)set break resume
)sys rm -f RealSolvePackage.output
)spool RealSolvePackage.output
)set message test on
\end{verbatim}
(--S 1 of 13
  p := 4*x^3 - 3*x^2 + 2*x - 4
  --R
  --R
  --R (1) 4x - 3x + 2x - 4
  --R Type: Polynomial(Integer)
--E 1

--S 2 of 13
ans1 := solve(p,0.01)$REALSOLVE
--R
--R (2) [1.11328125]
--R Type: List(Float)
--E 2

--S 3 of 13
ans2 := solve(p::POLY(FRAC(INT)),0.01)$REALSOLVE
--R
--R (3) [1.11328125]
--R Type: List(Float)
--E 3

--S 4 of 13
R := Integer
--R
--R (4) Integer
--R Type: Domain
--E 4

--S 5 of 13
ls : List Symbol := [x,y,z,t]
--R
--R (5) [x,y,z,t]
--R Type: List(Symbol)
--E 5

--S 6 of 13
ls2 : List Symbol := [x,y,z,t,new($)Symbol]
--R
--R (6) [x,y,z,t,%A]
--R Type: List(Symbol)
--E 6
pack := ZDSOLVE(R,ls,ls2)
--R
--R
--R (7) ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])
--R Type: Domain
--E 7

p1 := x**2*y*z + y*z
--R
--R
--R 2
--R (8) (x + 1)y z
--R Type: Polynomial(Integer)
--E 8

p2 := x**2*y**2*z + x + z
--R
--R
--R 2 2
--R (9) (x y + 1)z + x
--R Type: Polynomial(Integer)
--E 9

p3 := x**2*y**2*z**2 + z + 1
--R
--R
--R 2 2 2
--R (10) x y z + z + 1
--R Type: Polynomial(Integer)
--E 10

lp := [p1, p2, p3]
--R
--R
--R 2 2 2 2 2 2
--R (11) [(x + 1)y z,(x y + 1)z + x,x y z + z + 1]
--R Type: List(Polynomial(Integer))
--E 11

lsv::List(Symbol):=[x,y,z]
--R
--R
--R (12) [x,y,z]
This package provides numerical solutions of systems of polynomial equations for use in ACPLUT.

```plaintext
p := 4*x^3 - 3*x^2 + 2*x - 4
ans1 := solve(p,0.01)$REALSOLV
ans2 := solve(p::POLY(FRAC(INT)),0.01)$REALSOLV
R := Integer
ls : List Symbol := [x,y,z,t]
lhs : List Symbol := [x,y,z,t,new()$Symbol]
pack := ZDSOLVE(R,ls,lhs)
p1 := x**2*y*z + y*z
p2 := x**2*y**2*z + x + z
p3 := x**2*y**2*z**2 + z + 1
lp := [p1, p2, p3]
ans3 := realSolve(lp,[x,y,z],0.01)
```

See Also:
RealSolvePackage (REALSOLV)

Exports:
realSolve  solve

--- package REALSOLV RealSolvePackage ---

>abbrev package REALSOLV RealSolvePackage
++ Description:
++ This package provides numerical solutions of systems of
++ polynomial equations for use in ACPLST

RealSolvePackage(): Exports == Implementation where
  I ==> Integer
  IE ==> IndexedExponents Symbol
  L ==> List
  NF ==> Float
  P ==> Polynomial
  RN ==> Fraction Integer
  SE ==> Symbol
  RFI ==> Fraction Polynomial Integer
  LIFT ==> PolynomialCategoryLifting(IE,SE,RN,P RN, RFI)
  SOLV ==> FloatingRealPackage Float

Exports ==> with
  solve: (P RN,NF) -> L NF
  ++ solve(p,eps) finds the real zeroes of a
  ++ univariate rational polynomial p with precision eps.
++X p := 4*x^3 - 3*x^2 + 2*x - 4
++X solve(p::POLY(FRAC(INT)),0.01)$REALSOLV

solve: (P I,NF) -> L NF
++ solve(p,eps) finds the real zeroes of a univariate
++ integer polynomial p with precision eps.
++
++X p := 4*x^3 - 3*x^2 + 2*x - 4
++X solve(p,0.01)$REALSOLV

realSolve: (L P I,L SE,NF) -> L L NF
++ realSolve(lp,lv,eps) = compute the list of the real
++ solutions of the list lp of polynomials with integer
++ coefficients with respect to the variables in lv,
++ with precision eps.
++
++X p1 := x**2*y*z + y*z
++X p2 := x**2*y**2*z + x + z
++X p3 := x**2*y**2*z**2 + z + 1
++X lp := [p1, p2, p3]
++X realSolve(lp,[x,y,z],0.01)

Implementation ==> add

prn2rfi: P RN -> RFI
prn2rfi p ==
  map(x+->x::RFI, x+->(numer(x)::RFI)/(denom(x)::RFI), p)$LIFT

pi2rfi: P I -> RFI
pi2rfi p == p :: RFI

solve(p:P RN,eps:NF) == realRoots(prn2rfi p, eps)$SOLV

solve(p:P I,eps:NF) == realRoots(p::RFI, eps)$SOLV

realSolve(lp,lv,eps) ==
  realRoots(map(pi2rfi, lp)$ListFunctions2(P I,RFI),lv,eps)$SOLV
package REAL0 RealZeroPackage

— RealZeroPackage.input —

)set break resume
)sys rm -f RealZeroPackage.output
)spool RealZeroPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RealZeroPackage
--E 1

)spool
)lisp (bye)

———

— RealZeroPackage.help —

====================================================================
RealZeroPackage examples
====================================================================

This package provides functions for finding the real zeros of univariate polynomials over the integers to arbitrary user-specified precision. The results are returned as a list of isolating intervals which are expressed as records with "left" and "right" rational number components.

See Also:
o )show RealZeroPackage

———
RealZeroPackage (REAL0)

Exports:
midpoint midpoints realZeros refine

--- package REAL0 RealZeroPackage ---

)abbrev package REAL0 RealZeroPackage
++ Author: Andy Neff
++ Description:
++ This package provides functions for finding the real zeros
++ of univariate polynomials over the integers to arbitrary user-specified
++ precision. The results are returned as a list of
++ isolating intervals which are expressed as records with
++ "left" and "right" rational number components.

RealZeroPackage(Pol): T == C where
   Pol: UnivariatePolynomialCategory Integer
   RN ==> Fraction Integer
   Interval ==> Record(left : RN, right : RN)
   isoList ==> List(Interval)
   T == with
      -- next two functions find isolating intervals
      realZeros: (Pol) -> isoList
         ++ realZeros(pol) returns a list of isolating intervals for
         ++ all the real zeros of the univariate polynomial pol.
      realZeros: (Pol, Interval) -> isoList
         ++ realZeros(pol, range) returns a list of isolating intervals
         ++ for all the real zeros of the univariate polynomial pol which
         ++ lie in the interval expressed by the record range.
      -- next two functions return intervals smaller then tolerance
      realZeros: (Pol, RN) -> isoList
         ++ realZeros(pol, eps) returns a list of intervals of length less
         ++ than the rational number eps for all the real roots of the
         ++ polynomial pol.
      realZeros: (Pol, Interval, RN) -> isoList
         ++ realZeros(pol, int, eps) returns a list of intervals of length
CHAPTER 19. CHAPTER R

++ less than the rational number eps for all the real roots of the
++ polynomial pol which lie in the interval expressed by the
++ record int.
refine: (Pol, Interval, RN) -> Interval
++ refine(pol, int, eps) refines the interval int containing
++ exactly one root of the univariate polynomial pol to size less
++ than the rational number eps.
refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
++ refine(pol, int, range) takes a univariate polynomial pol and
++ and isolating interval int containing exactly one real
++ root of pol; the operation returns an isolating interval which
++ is contained within range, or "failed" if no such isolating interval exists.
midpoint: Interval -> RN
++ midpoint(int) returns the midpoint of the interval int.
midpoints: isoList -> List RN
++ midpoints(isolist) returns the list of midpoints for the list
++ of intervals isolist.
C == add
--Local Functions
makeSqfr: Pol -> Pol
ReZeroSqfr: (Pol) -> isoList
PosZero: (Pol) -> isoList
Zero1: (Pol) -> isoList
transMult: (Integer, Pol) -> Pol
transMultInv: (Integer, Pol) -> Pol
transAdd1: (Pol) -> Pol
invert: (Pol) -> Pol
minus: (Pol) -> Pol
negate: Interval -> Interval
rootBound: (Pol) -> Integer
var: (Pol) -> Integer

negate(int : Interval):Interval == [-int.right,-int.left]

midpoint(i : Interval):RN == (1/2)*(i.left + i.right)

midpoints(li : isoList) : List RN ==
[midpoint x for x in li]

makeSqfr(F : Pol):Pol ==
  sqfr := squareFree F
  F := */[s.factor for s in factors(sqfr)]

realZeros(F : Pol) ==
  ReZeroSqfr makeSqfr F

realZeros(F : Pol, rn : RN) ==
  F := makeSqfr F
  [refine(F,int,rn) for int in ReZeroSqfr(F)]
realZeros(F : Pol, bounds : Interval) ==
  F := makeSqfr F
  [rint::Interval for int in ReZeroSqfr(F) |
   (rint:=refine(F,int,bounds)) case Interval]

realZeros(F : Pol, bounds : Interval, rn : RN) ==
  F := makeSqfr F
  [refine(F,int,rn) for int in realZeros(F,bounds)]

ReZeroSqfr(F : Pol) ==
  F = 0 => error "ReZeroSqfr: zero polynomial"
  L : isoList := []
  degree(F) = 0 => L
  if (r := minimumDegree(F)) > 0 then
    L := [[0,0]$Interval]
    tempF := F exquo monomial(1, r)
    if not (tempF case "failed") then
      F := tempF
    J : isoList := [negate int for int in reverse(PosZero(minus(F)))]
    K : isoList := PosZero(F)
    append(append(J, L), K)

PosZero(F : Pol) == --F is square free, primitive
  --and F(0) ^= 0; returns isoList for positive
  --roots of F

  b : Integer := rootBound(F)
  F := transMult(b,F)
  L : isoList := Zero1(F)
  int : Interval
  L := [[b*int.left, b*int.right]$Interval for int in L]

Zero1(F : Pol) == --returns isoList for roots of F in (0,1)
  J : isoList
  K : isoList
  L : isoList
  L := []
  (v := var(transAdd1(invert(F)))) = 0 => []
  v = 1 => L := [[0,1]$Interval]
  G : Pol := transMultInv(2, F)
  H : Pol := transAdd1(G)
  if minimumDegree H > 0 then
    -- H has a root at 0 => F has one at 1/2, and G at 1
    L := [[1/2,1/2]$Interval]
    Q : Pol := monomial(1, 1)
    tempH : Union(Pol, "failed") := H exquo Q
    if not (tempH case "failed") then H := tempH
    Q := Q + monomial(-1, 0)
    tempG : Union(Pol, "failed") := G exquo Q
    if not (tempG case "failed") then G := tempG
int : Interval
J := [[(int.left+1) * (1/2), (int.right+1) * (1/2)]$Interval
   for int in Zero1(H)]
K := [[int.left * (1/2), int.right * (1/2)]$Interval
   for int in Zero1(G)]
append(append(J, L), K)

rootBound(F : Pol) == --returns power of 2 that is a bound
   --for the positive roots of F
   if leadingCoefficient(F) < 0 then F := -F
   lcoef := leadingCoefficient(F)
   F := reductum(F)
i : Integer := 0
while not (F = 0) repeat
   if (an := leadingCoefficient(F)) < 0 then i := i - an
   F := reductum(F)
b : Integer := 1
while (b * lcoef) <= i repeat
   b := 2 * b
   b

transMult(c : Integer, F : Pol) ==
   --computes Pol G such that G(x) = F(c*x)
   G : Pol := 0
   while not (F = 0) repeat
      n := degree(F)
      G := G + monomial((c**n) * leadingCoefficient(F), n)
      F := reductum(F)
G

transMultInv(c : Integer, F : Pol) ==
   --computes Pol G such that G(x) = (c**n) * F(x/c)
   d := degree(F)
   cc : Integer := 1
   G : Pol := monomial(leadingCoefficient F, d)
   while (F:=reductum(F)) ^= 0 repeat
      n := degree(F)
      cc := cc*(c**(d-n):NonNegativeInteger)
      G := G + monomial(cc * leadingCoefficient(F), n)
      d := n
   G

-- otransAdd1(F : Pol) ==
--   --computes Pol G such that G(x) = F(x+1)
--   G : Pol := F
--   n : Integer := 1
--   while (F := differentiate(F)) ^= 0 repeat
--      if not ((tempF := F exquo n) case "failed") then F := tempF
--      G := G + F
--      n := n + 1
transAdd1(F : Pol) ==
  -- computes Pol G such that G(x) = F(x+1)
  n := degree F
  v := vectorise(F, n+1)
  for i in 0..(n-1) repeat
    for j in (n-i)..n repeat
      qsetelt!_[v,j, qelt(v,j) + qelt(v,(j+1))]
  ans : Pol := 0
  for i in 0..n repeat
    ans := ans + monomial(qelt(v,(i+1)),i)
  ans

minus(F : Pol) ==
  -- computes Pol G such that G(x) = F(-x)
  G : Pol := 0
  while not (F = 0) repeat
    n := degree(F)
    coef := leadingCoefficient(F)
    odd? n =>
      G := G + monomial(-coef, n)
      F := reductum(F)
    G := G + monomial(coef, n)
    F := reductum(F)
  G

invert(F : Pol) ==
  -- computes Pol G such that G(x) = (x**n) * F(1/x)
  G : Pol := 0
  n := degree(F)
  while not (F = 0) repeat
    G := G + monomial(leadingCoefficient(F),
      (n-degree(F))::NonNegativeInteger)
    F := reductum(F)
  G

var(F : Pol) ==
  -- number of sign variations in coefs of F
  i : Integer := 0
  LastCoef : Boolean
  next : Boolean
  LastCoef := leadingCoefficient(F) < 0
  while not ((F := reductum(F)) = 0) repeat
    next := leadingCoefficient(F) < 0
    if ((not LastCoef) and next) or
      ((not next) and LastCoef) then i := i+1
    LastCoef := next
  i
refine(F : Pol, int : Interval, bounds : Interval) ==
  lseg := min(int.right, bounds.right) - max(int.left, bounds.left)
  lseg < 0 => "failed"
  lseg = 0 =>
    pt :=
      int.left = bounds.right => int.left
      int.right
    elt(transMultInv(denom(pt), F), numer pt) = 0 => [pt, pt]
    "failed"
  lseg = int.right - int.left => int
  refine(F, refine(F, int, lseg), bounds)

refine(F : Pol, int : Interval, eps : RN) ==
  a := int.left
  b := int.right
  a\=b => [a, b] Interval
  an : Integer := numer(a)
  ad : Integer := denom(a)
  bn : Integer := numer(b)
  bd : Integer := denom(b)
  xfl : Boolean := false
  if (u:=elt(transMultInv(ad, F), an)) = 0 then
    F := (F exquo (monomial(ad, 1) - monomial(an, 0)))::Pol
    u:=elt(transMultInv(ad, F), an)
  if (v:=elt(transMultInv(bd, F), bn)) = 0 then
    F := (F exquo (monomial(bd, 1) - monomial(bn, 0)))::Pol
    v:=elt(transMultInv(bd, F), bn)
    u:=elt(transMultInv(ad, F), an)
  if u > 0 then (F:=-F; v:=-v)
  if v < 0 then
    error [int, "is not a valid isolation interval for", F]
  if eps <= 0 then error "precision must be positive"
  while (b - a) >= eps repeat
    mid : RN := (b + a) * (1/2)
    midn : Integer := numer(mid)
    midd : Integer := denom(mid)
    (v := elt(transMultInv(midd, F), midn)) < 0 =>
      a := mid
      an := midn
      ad := midd
      v > 0 =>
        b := mid
        bn := midn
        bd := midd
      v = 0 =>
        a := mid
        b := mid
        an := midn
        ad := midd
      xfl := true
This package provides functions for finding the real zeros of univariate polynomials over the rational numbers to arbitrary user-specified precision. The results are returned as a list of isolating intervals, expressed as records with "left" and "right" rational number components.
RealZeroPackageQ (REAL0Q)

Exports:
realZeros refine

— package REAL0Q RealZeroPackageQ —

)abbrev package REAL0Q RealZeroPackageQ
++ Author: Andy Neff, Barry Trager
++ Date Last Updated: 7 April 1991
++ Description:
++ This package provides functions for finding the real zeros of univariate
++ polynomials over the rational numbers to arbitrary user-specified
++ precision. The results are returned as a list of isolating intervals,
++ expressed as records with "left" and "right" rational number components.

RealZeroPackageQ(Pol): T == C where
RN ==> Fraction Integer
I ==> Integer
SUP ==> SparseUnivariatePolynomial
Pol: UnivariatePolynomialCategory RN
Interval ==> Record(left : RN, right : RN)
isoList ==> List(Interval)
ApproxInfo ==> Record(approx : RN, exFlag : Boolean)
T == with
-- next two functions find isolating intervals
realZeros: (Pol) -> isoList
++ realZeros(pol) returns a list of isolating intervals for
++ all the real zeros of the univariate polynomial pol.
realZeros: (Pol, Interval) -> isoList
  ++ realZeros(pol, range) returns a list of isolating intervals
  ++ for all the real zeros of the univariate polynomial pol which
  ++ lie in the interval expressed by the record range.
-- next two functions return intervals smaller then tolerance
realZeros: (Pol, RN) -> isoList
  ++ realZeros(pol, eps) returns a list of intervals of length less
  ++ than the rational number eps for all the real roots of the
  ++ polynomial pol.
realZeros: (Pol, Interval, RN) -> isoList
  ++ realZeros(pol, int, eps) returns a list of intervals of length
  ++ less than the rational number eps for all the real roots of the
  ++ polynomial pol which lie in the interval expressed by the
  ++ record int.
refine: (Pol, Interval, RN) -> Interval
  ++ refine(pol, int, eps) refines the interval int containing
  ++ exactly one root of the univariate polynomial pol to size less
  ++ than the rational number eps.
refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
  ++ refine(pol, int, range) takes a univariate polynomial pol and
  ++ and isolating interval int which must contain exactly one real
  ++ root of pol, and returns an isolating interval which
  ++ is contained within range, or "failed" if no such isolating interval exists.

C == add
  import RealZeroPackage SparseUnivariatePolynomial Integer

convert2PolInt: Pol -> SparseUnivariatePolynomial Integer

convert2PolInt(f : Pol) ==
  pden:I := lcm([denom c for c in coefficients f])
  map(numer,pden * f)$UnivariatePolynomialCategoryFunctions2(RN,Pol,I,SUP I)

realZeros(f : Pol) == realZeros(convert2PolInt f)
realZeros(f : Pol, rn : RN) == realZeros(convert2PolInt f, rn)
realZeros(f : Pol, bounds : Interval) ==
  realZeros(convert2PolInt f, bounds)
realZeros(f : Pol, bounds : Interval, rn : RN) ==
  realZeros(convert2PolInt f, bounds, rn)
refine(f : Pol, int : Interval, eps : RN) ==
  refine(convert2PolInt f, int, eps)
refine(f : Pol, int : Interval, bounds : Interval) ==
  refine(convert2PolInt f, int, bounds)
package RMCAT2 RectangularMatrixCategoryFunctions2

RectangularMatrixCategoryFunctions2 examples

RectangularMatrixCategoryFunctions2 provides functions between two matrix domains. The functions provided are map and reduce.

See Also:
o  )show RectangularMatrixCategoryFunctions2
Exports:

map  reduce

— package RMCAT2 RectangularMatrixCategoryFunctions2 —

)abbrev package RMCAT2 RectangularMatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 12 June 1991
++ Description:
++ \spadtype{RectangularMatrixCategoryFunctions2} provides functions between
++ two matrix domains. The functions provided are \spadfun{map} and
++ \spadfun{reduce}.

RectangularMatrixCategoryFunctions2(m,n,R1,Row1,Col1,M1,R2,Row2,Col2,M2):_
    Exports == Implementation where
    m,n : NonNegativeInteger
    R1 : Ring
    Row1 : DirectProductCategory(n, R1)
    Col1 : DirectProductCategory(m, R1)
    M1 : RectangularMatrixCategory(m,n,R1,Row1,Col1)
    R2 : Ring
    Row2 : DirectProductCategory(n, R2)
    Col2 : DirectProductCategory(m, R2)
    M2 : RectangularMatrixCategory(m,n,R2,Row2,Col2)

Exports ==> with
    map: (R1 -> R2,M1) -> M2
    ++ \spad{map(f,m)} applies the function \spad{f} to the elements of the
    ++ matrix \spad{m}.
    reduce: (R1,R2) -> R2,M1,R2) -> R2
    ++ \spad{reduce(f,m,r)} returns a matrix \spad{n} where
    ++ \spad{n[i,j] = f(m[i,j],r)} for all indices spad{i} and \spad{j}.

Implementation ==> add
CHAPTER 19. CHAPTER R

\[ \text{minr} \Rightarrow \text{minRowIndex} \]
\[ \text{maxr} \Rightarrow \text{maxRowIndex} \]
\[ \text{minc} \Rightarrow \text{minColIndex} \]
\[ \text{maxc} \Rightarrow \text{maxColIndex} \]

\[
\text{map}(f, \text{mat}) =
\]
\[ \text{ans} : M2 := \text{new}(m,n,0) \text{Matrix}(R2) \text{ pretend } M2 \]
\[ \text{for } i \text{ in minr(mat..maxr(mat) for } k \text{ in minr(ans..maxr(ans) repeat} \]
\[ \text{for } j \text{ in minc(mat..maxc(mat) for } l \text{ in minc(ans..maxc(ans) repeat} \]
\[ \text{qsetelt_!}(\text{ans pretend Matrix } R2,k,l,f \text{ qelt(mat,i,j)}) \]
\[ \text{ans} \]

\[
\text{reduce}(f, \text{mat}, \text{ident}) =
\]
\[ s := \text{ident} \]
\[ \text{for } i \text{ in minr(mat..maxr(mat) repeat} \]
\[ \text{for } j \text{ in minc(mat..maxc(mat) repeat} \]
\[ s := f(\text{qelt(mat,i,j)},s) \]
\[ s \]

———

— RMCAT2.dotabb ——

"RMCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RMCAT2"]
"RMATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RMATCAT"]
"RMCAT2" -> "RMATCAT"

———

package RECOP RecurrenceOperator

—— RecurrenceOperator.input ——

)set break resume
)sys rm -f RecurrenceOperator.output
)spool RecurrenceOperator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RecurrenceOperator
--E 1
RecurrenceOperator (RECOP)

Exports:
getEq    getOp    evalADE    evalRec
getShiftRec  numberOfValuesNeeded  shiftInfoRec

--- package RECOP RecurrenceOperator ---

)abbrev package RECOP RecurrenceOperator
++ Author: Martin Rubey
++ Description:
++ This package provides an operator for the n-th term of a recurrence and an
++ operator for the coefficient of x^n in a function specified by a functional
RecurrenceOperator(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
  F: Join(FunctionSpace R, AbelianMonoid, RetractableTo Integer, 
      RetractableTo Symbol, PartialDifferentialRing Symbol, 
      CombinatorialOpsCategory)
--RecurrenceOperator(F): Exports == Implementation where
-- F: Join(ExpressionSpace, AbelianMonoid, RetractableTo Integer, 
--      RetractableTo Symbol, PartialDifferentialRing Symbol)

Exports == with

  evalRec: (BasicOperator, Symbol, F, F, F, List F) -> F
  \spad{evalRec(u, dummy, n, n0, eq, values)} creates an expression that 
  \spad{+} stands for \spad{u(n0)}, where \spad{u(n)} is given by the equation \spad{eq}. However, for 
  \spad{+} technical reasons the variable \spad{n} has to be replaced by a dummy 
  \spad{+} variable \spad{dummy} in \spad{eq}. The argument \spad{values} specifies the initial values 
  \spad{+} of the recurrence \spad{u(0)}, \spad{u(1)},... 
  \spad{+} For the moment we don't allow recursions that contain \spad{u} inside of 
  \spad{+} another operator.

  evalADE: (BasicOperator, Symbol, F, F, F, List F) -> F
  \spad{evalADE(f, dummy, x, n, eq, values)} creates an expression that 
  \spad{+} stands for the coefficient of \spad{x^n} in the Taylor expansion of \spad{f(x)}, 
  \spad{+} where \spad{f(x)} is given by the functional equation \spad{eq}. However, for 
  \spad{+} technical reasons the variable \spad{x} has to be replaced by a dummy 
  \spad{+} variable \spad{dummy} in \spad{eq}. The argument \spad{values} specifies the first few 
  \spad{+} Taylor coefficients.

  getEq: F -> F
  \spad{getEq f} returns the defining equation, if \spad{f} represents the 
  \spad{+} coefficient of an ADE or a recurrence.

  getOp: F -> BasicOperator
  \spad{getOp f}, if \spad{f} represents the coefficient of a recurrence or 
  \spad{+} ADE, returns the operator representing the solution

-- should be local
  numberOfValuesNeeded: (Integer, BasicOperator, Symbol, F) -> Integer

-- should be local
  if R has Ring then
    getShiftRec: (BasicOperator, Kernel F, Symbol) -> Union(Integer, "failed")

shiftInfoRec: (BasicOperator, Symbol, F) ->
  Record(max: Union(Integer, "failed"), 
  ord: Union(Integer, "failed"), 
  ker: Kernel F)
Defining new operators

We define two new operators, one for recurrences, the other for functional equations. The operators for recurrences represents the $n^{th}$ term of the corresponding sequence, the other the coefficient of $x^n$ in the Taylor series expansion.

---

```plaintext
implementation: RecurrenceOperator

oprecur := operator("rootOfRec":Symbol)$BasicOperator
opADE := operator("rootOfADE":Symbol)$BasicOperator
setProperty(oprecur, "%dummyVar", 2 pretend None)
setProperty(opADE, "%dummyVar", 2 pretend None)
```

Setting these properties implies that the second and third arguments of oprecure are dummy variables and affects `tower$ES`: the second argument will not appear in `tower$ES`, if it does not appear in any argument but the first and second. The third argument will not appear in `tower$ES`, if it does not appear in any other argument. (%defsum is a good example)

The arguments of the two operators are as follows:

1. eq, i.e. the vanishing expression

```plaintext
---

implementation: RecurrenceOperator

eqAsF: List F -> F
eqAsF l == l.1
```

2. dummy, a dummy variable to make substitutions possible

```plaintext
---

implementation: RecurrenceOperator

dummy: List F -> Symbol
dummy l == retract(l.2)@Symbol
dummyAsF: List F -> F
dummyAsF l == 1.2
```
3. the variable for display

<table>
<thead>
<tr>
<th>implementation: RecurrenceOperator</th>
</tr>
</thead>
<tbody>
<tr>
<td>displayVariable: List F → F</td>
</tr>
<tr>
<td>displayVariable l == 1.3</td>
</tr>
</tbody>
</table>

4. operatorName(argument)

<table>
<thead>
<tr>
<th>implementation: RecurrenceOperator</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorName: List F → BasicOperator</td>
</tr>
<tr>
<td>operatorName l == operator(kernels(1.4).1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operatorNameAsF: List F → F</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorNameAsF l == 1.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operatorArgument: List F → F</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorArgument l == argument(kernels(1.4).1).1</td>
</tr>
</tbody>
</table>

Concerning rootOfADE, note that although we have arg as argument of the operator, it is intended to indicate the coefficient, not the argument of the power series.

5. values in reversed order.

- rootOfRec: maybe values should be preceded by the index of the first given value. Currently, the last value is interpreted as \( f(0) \).
- rootOfADE: values are the first few coefficients of the power series expansion in order.

<table>
<thead>
<tr>
<th>implementation: RecurrenceOperator</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialValues: List F → List F</td>
</tr>
<tr>
<td>initialValues l == rest(l, 4)</td>
</tr>
</tbody>
</table>
Recurrences

Extracting some information from the recurrence

We need to find out whether we can determine the next term of the sequence, and how many initial values are necessary.

— implementation: RecurrenceOperator —

```lisp
if R has Ring then
getShiftRec(op: BasicOperator, f: Kernel F, n: Symbol)
  : Union(Integer, "failed") ==
a := argument f
if every?(z -> freeOf?(z, n::F), a) then return 0
if #a ~= 1 then error "RECOP: operator should have only one argument"
p := univariate(a.1, retract(n::F)@Kernel(F))
if denominator p ~= 1 then return "failed"
num := numer p
if degree num = 1 and coefficient(num, 1) = 1
  and every?(z -> freeOf?(z, n::F), coefficients num)
then return retractIfCan(coefficient(num, 0))
else return "failed"
```

-- if the recurrence is of the form
-- $p(n, f(n+m-o), f(n+m-o+1), \ldots, f(n+m)) = 0$
-- in which case shiftInfoRec returns [m, o, f(n+m)].

```lisp
shiftInfoRec(op: BasicOperator, argsym: Symbol, eq: F):
  Record(max: Union(Integer, "failed"),
         ord: Union(Integer, "failed"),
         ker: Kernel F) ==
```

-- ord and ker are valid only if all shifts are Integers
-- ker is the kernel of the maximal shift.
maxShift: Integer
minShift: Integer
nextKernel: Kernel F

-- We consider only those kernels that have op as operator. If there is none,
-- we raise an error. For the moment we don't allow recursions that contain op
-- inside of another operator.
error? := true

for f in kernels eq repeat
if is?(f, op) then
    shift := getShiftRec(op, f, argsym)
if error? then
    error? := false
nextKernel := f
if shift case Integer then
    maxShift := shift
    minShift := shift
else return ["failed", "failed", nextKernel]
else
    if shift case Integer then
        if maxShift < shift then
            maxShift := shift
            nextKernel := f
        if minShift > shift then
            minShift := shift
    else return ["failed", "failed", nextKernel]
if error? then error "evalRec: equation does not contain operator"
[maxShift, maxShift - minShift, nextKernel]

Evaluating a recurrence

— implementation: RecurrenceOperator —

evalRec(op, argsym, argdisp, arg, eq, values) ==
    if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
or (n < 0)
then
    shiftInfo := shiftInfoRec(op, argsym, eq)
    if (shiftInfo.ord case "failed") or ((shiftInfo.ord)::Integer > 0)
then
    kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))
else
    p := univariate(eq, shiftInfo.ker)
    num := numer p

    -- If the degree is 1, we can return the function explicitly.

    if degree num = 1 then
        eval(-coefficient(num, 0)/coefficient(num, 1), argsym::F,
            arg::F-(shiftInfo.max)::Integer::F)
    else
kernel(oprecur,
    append([eq, argsym::F, argdisp, op(arg)], values))

else
    len: Integer := #values
    if n < len
        then values.(len-n)
    else
        shiftInfo := shiftInfoRec(op, argsym, eq)

        if shiftInfo.max case Integer then
            p := univariate(eq, shiftInfo.ker)

            num := numer p

            if degree num = 1 then
                next := -coefficient(num, 0)/coefficient(num, 1)
                nextval := eval(next, argsym::F, 
                    (len-(shiftInfo.max)::Integer)::F)
                newval := eval(nextval, op, 
                    z->evalRec(op, argsym, argdisp, z, eq, values))
                evalRec(op, argsym, argdisp, arg, eq, cons(newval, values))
            else
                kernel(oprecur,
                    append([eq, argsym::F, argdisp, op(arg)], values))
            end if
        else
            kernel(oprecur,
                append([eq, argsym::F, argdisp, op(arg)], values))
        end if
    end if

numberOfValuesNeeded(numberOfValues: Integer,
    op: BasicOperator,
    argsym: Symbol, eq: F): Integer ==

    order := shiftInfoRec(op, argsym, eq).ord
    if order case Integer
        then min(numberOfValues, retract(order)@Integer)
    else numberOfValues

else
    evalRec(op, argsym, argdisp, arg, eq, values) ==
    kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))

numberOfValuesNeeded(numberOfValues: Integer,
    op: BasicOperator, argsym: Symbol, eq: F): Integer ==

    numberOfValues
Setting the evaluation property of \texttt{oprecur}

\texttt{irecur} is just a wrapper that allows us to write a recurrence relation as an operator.

\begin{verbatim}
    — implementation: RecurrenceOperator —

    irecur: List F -> F
    irecur l ==
        evalRec(operatorName l, dummy l, displayVariable l, operatorArgument l, eqAsF l, initialValues l)
        evaluate(oprecur, irecur)$BasicOperatorFunctions1(F)

    ————

    Displaying a recurrence relation

    — implementation: RecurrenceOperator —

    ddrec: List F -> OutputForm
    ddrec l ==
        op := operatorName l
        values := reverse l
        eq := eqAsF l

        numberOfValues := numberOfValuesNeeded(#values-4, op, dummy l, eq)

        vals: List OutputForm
            := cons(eval(eq, dummyAsF l, displayVariable l)::OutputForm = _
              0::OutputForm,
              [elt(op::OutputForm, [(i-1)::OutputForm]) = _
               (values.i)::OutputForm
               for i in 1..numberOfValues])

        bracket(hconcat([((operatorNameAsF l)::OutputForm,
                          ": ",
                          commaSeparate vals)]))

        setProperty(oprecur, "%specialDisp",
                     ddrec@(List F -> OutputForm) pretend None)

    ————
\end{verbatim}
Functional Equations

Determining the number of initial values for ADE’s

We use Joris van der Hoeven’s instructions for ADE’s. Given $Q = p(f, f', \ldots, f^{(r)})$ we first need to differentiate $Q$ with respect to $f^{(i)}$ for $i \in \{0, 1, \ldots, r\}$, plug in the given truncated power series solution and determine the valuation.

— NOTYET implementation: RecurrenceOperator —

```
getValuation(op, argsym, eq, maxorder, values): Integer ==
  max: Integer := -1;
  ker: Kernel F
  for i in 0..maxorder repeat
    ker := D(op(argsym), argsym, i)::Kernel F
    pol := univariate(eq, ker)
    dif := D pol
    ground numer(dif.D(op(argsym), argsym, i))
```

Extracting some information from the functional equation

getOrder returns the maximum derivative of op occurring in f.

— implementation: RecurrenceOperator —

```
getOrder(op: BasicOperator, f: Kernel F): NonNegativeInteger ==
  res: NonNegativeInteger := 0
  g := f
  while is?(g, %diff) repeat
    g := kernels(argument(g).1).1
    res := res+1
  
  if is?(g, op) then res else 0
```

Extracting a coefficient given a functional equation

— implementation: RecurrenceOperator —

```
evalADE(op, argsym, argdisp, arg, eq, values) ==
  if not freeOf?(eq, retract(argdisp)@Symbol)
```
then error "RECOP: The argument should not be used in the equation of the ADE"

if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed") then
  -- try to determine the necessary number of initial values
  keq := kernels eq
  order := getOrder(op, keq.1)
  for k in rest keq repeat order := max(order, getOrder(op, k))

  p: Fraction SparseUnivariatePolynomial F := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F

  if one? degree numer p
    -- the equation is holonomic
    then kernel(opADE, 
      append([eq, argsym::F, argdisp, op(arg)], reverse first(reverse values, order)))
    else kernel(opADE, 
      append([eq, argsym::F, argdisp, op(arg)], values))
  else
    if n < 0
      then 0
    else
      keq := kernels eq
      order := getOrder(op, keq.1)
      -- output(hconcat("The order is ", order::OutputForm))$OutputPackage
      for k in rest keq repeat order := max(order, getOrder(op, k))

      p: Fraction SparseUnivariatePolynomial F := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F

      -- output(hconcat("p: ", p::OutputForm))$OutputPackage

      if degree numer p > 1
        then
          -- kernel(opADE, 
          -- append([eq, argsym::F, argdisp, op(arg)], values))
          s := seriesSolve(eq, op, argsym, reverse values)
          $ExpressionSolve(R, F, UnivariateFormalPowerSeries F, UnivariateFormalPowerSeries SparseUnivariatePolynomialExpressions F)
          elt(s, n::Integer::NonNegativeInteger)
        else
          s := seriesSolve(eq, op, argsym, first(reverse values, order))
          $ExpressionSolve(R, F,
elt(s, n::Integer::NonNegativeInteger)

iADE: List F -> F
-- This is just a wrapper that allows us to write a recurrence relation as an
-- operator.
  iADE l ==
    evalADE(operatorName l,
            dummy l, displayVariable l,
            operatorArgument l, eqAsF l, initialValues l)

evaluate(opADE, iADE)$BasicOperatorFunctions1(F)

getEq(f: F): F ==
  ker := kernels f
  if one?(#ker) and_
    (is?(operator(ker.1), "rootOfADE"::Symbol) or _
     is?(operator(ker.1), "rootOfRec"::Symbol)) then
    l := argument(ker.1)
    eval(eqAsF l, dummyAsF l, displayVariable l)
  else
    error "getEq: argument should be a single rootOfADE or rootOfRec object"

getOp(f: F): BasicOperator ==
  ker := kernels f
  if one?(#ker) and_
    (is?(operator(ker.1), "rootOfADE"::Symbol) or _
     is?(operator(ker.1), "rootOfRec"::Symbol)) then
    operatorName argument(ker.1)
  else
    error "getOp: argument should be a single rootOfADE or rootOfRec object"

Displaying a functional equation

— implementation: RecurrenceOperator —

ddADE: List F -> OutputForm
ddADE l ==
  op := operatorName l
  values := reverse l
vals: List OutputForm
   := cons(eval(eqAsFl, dummyAsFl, displayVariable l)::OutputForm = _
       0::OutputForm,
       [eval(D(op(dummyAsFl l), dummy l, i), _
         dummyAsFl l=0)::OutputForm = _
       (values.(i+1))::OutputForm * _
       factorial(box(i::R::F)$F)::OutputForm _
       for i in 0..min(4,#values-5)])

bracket(hconcat([bracket((displayVariable l)::OutputForm ** _
                    (operatorArgument l)::OutputForm),
                    (op(displayVariable l))::OutputForm, ":",
                    commaSeparate vals]))

setProperty(opADE, "%specialDisp",
            ddADE@(List F -> OutputForm) pretend None)

---

— RECOP.dotabb —

"RECOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RECOP"]
"EXPRSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRSOL"]
"RECOP" -> "EXPRSOL"

---

package RDIV ReducedDivisor

--- ReducedDivisor.input ---

)set break resume
)sys rm -f ReducedDivisor.output
)spool ReducedDivisor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ReducedDivisor
--E 1

)spool
)lisp (bye)
Exports:
order

--- package RDIV ReducedDivisor ---

)abbrev package RDIV ReducedDivisor
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 8 November 1994
++ Description:
++ Finds the order of a divisor over a finite field

ReducedDivisor(F1, UP, UPUP, R, F2): Exports == Implementation where
  F1       : Field
  UP       : UnivariatePolynomialCategory F1
  UPUP     : UnivariatePolynomialCategory Fraction UP
\textbf{R} : \text{FunctionFieldCategory}(F1, \text{UP}, \text{UPUP})

\textbf{F2} : \text{Join}(\text{Finite}, \text{Field})

\textbf{N} \implies \text{NonNegativeInteger}

\textbf{FD} \implies \text{FiniteDivisor}(F1, \text{UP}, \text{UPUP}, \text{R})

\textbf{UP2} \implies \text{SparseUnivariatePolynomial} \text{F2}

\textbf{UPUP2} \implies \text{SparseUnivariatePolynomial \text{Fraction} \text{UP2}}

\textbf{Exports} \implies \text{with}

\text{order}: (\text{FD}, \text{UPUP}, \text{F1} \to \text{F2}) \to \text{N}

++ \text{order}(f, u, g) \text{ \text{\textbackslash undocumented}}

\textbf{Implementation} \implies \text{add}

\textbf{algOrder} : (\text{FD}, \text{UPUP}, \text{F1} \to \text{F2}) \to \text{N}

\textbf{rootOrder}: (\text{FD}, \text{UP}, \text{N}, \text{F1} \to \text{F2}) \to \text{N}

\text{-- pp is not necessarily monic}

\text{order}(d, \text{pp}, f) ==

(r := \text{retractIfCan}(\text{reductum} \text{pp})\text{\textbackslash Union} (\text{Fraction} \text{UP}, "failed"))

\text{case} "failed" \implies \text{algOrder}(d, \text{pp}, f)

\text{rootOrder}(d, - \text{retract}(r::\text{Fraction}(\text{UP}) / \text{leadingCoefficient} \text{pp})\text{\textbackslash UP},

\text{degree} \text{pp}, f)

\text{algOrder}(d, \text{modulus}, \text{reduce}) ==

\text{redmod} := \text{map} (\text{reduce}, \text{modulus}) \text{\$MultipleMap} (F1, \text{UP}, \text{UPUP}, F2, \text{UP2}, \text{UPUP2})

\text{curve} := \text{AlgebraicFunctionField} (F2, \text{UP2}, \text{UPUP2}, \text{redmod})

\text{order}(\text{map} (\text{reduce},

d)\text{\$FiniteDivisorFunctions2} (F1, \text{UP}, \text{UPUP}, R, \text{F2}, \text{UP2}, \text{UPUP2}, \text{curve})

) \text{\$FindOrderFinite} (F2, \text{UP2}, \text{UPUP2}, \text{curve})

\text{rootOrder}(d, \text{radicand}, n, \text{reduce}) ==

\text{redrad} := \text{map} (\text{reduce},

\text{radicand})\text{\$UnivariatePolynomialCategoryFunctions2} (F1, \text{UP}, \text{F2}, \text{UP2})

\text{curve} := \text{RadicalFunctionField} (F2, \text{UP2}, \text{UPUP2}, \text{redrad)::\text{Fraction} \text{UP2}, n)

\text{order}(\text{map} (\text{reduce},

d)\text{\$FiniteDivisorFunctions2} (F1, \text{UP}, \text{UPUP}, R, \text{F2}, \text{UP2}, \text{UPUP2}, \text{curve})

) \text{\$FindOrderFinite} (F2, \text{UP2}, \text{UPUP2}, \text{curve})

\text{-- RDIV.dotabb --}

"RDIV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDIV"]

"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]

"RDIV" \to "FFCAT"
package ODERED ReduceLODE

— ReduceLODE.input —

)set break resume
)sys rm -f ReduceLODE.output
)spool ReduceLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ReduceLODE
--E 1

)spool
)lisp (bye)

———

— ReduceLODE.help —

====================================================================
ReduceLODE examples
====================================================================

Elimination of an algebraic from the coefficients of a linear ordinary differential equation.

See Also:
o )show ReduceLODE

———
ReduceLODE (ODERED)

Exports:
reduceLODE

— package ODERED ReduceLODE —

)abbrev package ODERED ReduceLODE
++ Author: Manuel Bronstein
++ Date Created: 19 August 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ Elimination of an algebraic from the coefficients
++ of a linear ordinary differential equation.

ReduceLODE(F, L, UP, A, LO): Exports == Implementation where
F : Field
L : LinearOrdinaryDifferentialOperatorCategory F
UP: UnivariatePolynomialCategory F
A : MonogenicAlgebra(F, UP)
LO: LinearOrdinaryDifferentialOperatorCategory A

V ==> Vector F
M ==> Matrix L

Exports ==> with
reduceLODE: (LO, A) -> Record(mat:M, vec:V)

Implementation ==> add
matF2L: Matrix F -> M
diff := D()$L

-- coerces a matrix of elements of F into a matrix of (order 0) L.O.D.O's
matF2L m ==
    map((f1:F):L+->f1::L, m)$MatrixCategoryFunctions2(F, V, V, Matrix F, L, Vector L, Vector L, M)

-- This follows the algorithm and notation of
-- "The Risch Differential Equation on an Algebraic Curve", M. Bronstein,
reduceLODE(l, g) ==
    n := rank()$A
-- md is the basic differential matrix (D x I + Dy)
    md := matF2L transpose derivationCoordinates(basis(), (f1:F):F+->diff f1)
    for i in minRowIndex md .. maxRowIndex md
        for j in minColIndex md .. maxColIndex md repeat
            md(i, j) := diff + md(i, j)
-- mdi will go through the successive powers of md
    mdi := copy md
    sys := matF2L(transpose regularRepresentation coefficient(l, 0))
    for i in 1..degree l repeat
        sys := sys +
            matF2L(transpose regularRepresentation coefficient(l, i)) * mdi
    mdi := md * mdi
    [sys, coordinates g]
ReductionOfOrder provides functions for reducing the order of linear ordinary differential equations once some solutions are known.

See Also:
  o )show ReductionOfOrder

Exports:
ReduceOrder

— ReductionOfOrder.help —


\text{ReductionOfOrder}(F, L): \text{Exports} \rightarrow \text{Impl}

\text{Exports} \rightarrow \text{with}
\text{ReduceOrder}: (L, F) \rightarrow L
++ \text{ReduceOrder}(op, s) \text{ returns } \text{op1} \text{ such that for any solution}
++ \text{op1}(z) \text{ of } \text{op1}(z = 0), \text{spad}(y = s \int z) \text{ is a solution of}
++ \text{op1}(op y = 0). \text{spad}(s) \text{ must satisfy } \text{spad}(op s = 0).

\text{ReduceOrder}: (L, \text{List} F) \rightarrow \text{Record(eq:L, op:List F)}
++ \text{ReduceOrder}(op, [f1, ..., fk]) \text{ returns } \text{op1}, [g1, ..., gk] \text{ such that}
++ \text{for any solution } \text{op1}(z) \text{ of } \text{op1}(op z = 0),
++ \text{spad}(y = gk \int (g_{k-1} \int (...) \int (g1 \int z) ...)) \text{ is a solution}
++ \text{of } \text{op1}(op y = 0). \text{Each } \text{spad}(fi) \text{ must satisfy } \text{spad}(op fi = 0).

\text{Impl} \rightarrow \text{add}
\text{ithcoef}: (L, Z, A) \rightarrow F
\text{locals}: (A, Z, Z) \rightarrow F
\text{localbinom}: (Z, Z) \rightarrow Z
\text{diff} := D() \in L

\text{localbinom}(j, i) \equiv (j > i \Rightarrow \text{binomial}(j, i+1); 0)
\text{locals}(s, j, i) \equiv (j > i \Rightarrow \text{qelt}(s, j - i - 1); 0)

\text{ReduceOrder}(1:L, \text{sols:List} F) \equiv
\text{empty?} \text{sols} \Rightarrow [1, \text{empty}]$
\text{neweq} := \text{ReduceOrder}(1, \text{sol := first} \text{sols})
\text{rec} := \text{ReduceOrder}(\text{neweq}, [\text{diff}(s / \text{sol}) \text{ for } s \text{ in rest} \text{sols}])
[\text{rec.eq}, \text{concat_!}(\text{rec.op}, \text{sol})]

\text{ithcoef}(eq, i, s) \equiv
\text{ans} := 0$
\text{while} eq = 0 \text{repeat}
\text{j} := \text{degree} eq$
\text{ans} := \text{ans} + \text{localbinom}(j, i) \ast \text{locals}(s, j, i) \ast \text{leadingCoefficient} eq$
\text{eq} := \text{reductum} eq
\text{ans}
CHAPTER 19. CHAPTER R

ReduceOrder(eq:L, sol:F) ==
  s:A := new(n := degree eq, 0)  -- will contain derivatives of sol
  si := sol                    -- will run through the derivatives
  qsetelt_!(s, 0, si)
  for i in 1..(n-1)::NonNegativeInteger repeat
    qsetelt_!(s, i, si := diff si)
  ans:L := 0
  for i in 0..(n-1)::NonNegativeInteger repeat
    ans := ans + monomial(ithcoef(eq, i, s), i)
  ans

— REDORDER.dotabb —

"REDORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REDORDER"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"REDORDER" -> "OREPCAT"
"REDORDER" -> "A1AGG"

———

package RSDCMPK RegularSetDecompositionPackage

—— RegularSetDecompositionPackage.input ——

)set break resume
)sys rm -f RegularSetDecompositionPackage.output
)spool RegularSetDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RegularSetDecompositionPackage
--E 1

)spool
)lisp (bye)

———
A package providing a new algorithm for solving polynomial systems by means of regular chains. Two ways of solving are proposed: in the sense of Zariski closure (like in Kalkbrener's algorithm) or in the sense of the regular zeros (like in Wu, Wang or Lazard methods). This algorithm is valid for any type of regular set. It does not care about the way a polynomial is added in an regular set, or how two quasi-components are compared (by an inclusion-test), or how the invertibility test is made in the tower of simple extensions associated with a regular set.

These operations are realized respectively by the domain TS and the packages QCMPACK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS).

The same way it does not care about the way univariate polynomial gcd (with coefficients in the tower of simple extensions associated with a regular set) are computed. The only requirement is that these gcd need to have invertible initials (normalized or not).

WARNING. There is no need for a user to call directly any operation of this package since they can be accessed by the domain TS. Thus, the operations of this package are not documented.

See Also:
- )show RegularSetDecompositionPackage
RegularSetDecompositionPackage (RSDCMPK)

Exports:
KrullNumber  algebraicDecompose  convert  decompose
internalDecompose  numberOfVariables  printInfo  transcendentalDecompose
upDateBranches

— package RSDCMPK RegularSetDecompositionPackage —

)abbrev package RSDCMPK RegularSetDecompositionPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/16/1998
++ Date Last Updated: 12/16/1998
++ References :
++ Description:
++ A package providing a new algorithm for solving polynomial systems
++ by means of regular chains. Two ways of solving are proposed:
++ in the sense of Zariski closure (like in Kalkbrener’s algorithm)
++ or in the sense of the regular zeros (like in Wu, Wang or Lazard
++ methods). This algorithm is valid for any type
++ of regular set. It does not care about the way a polynomial is
++ added in an regular set, or how two quasi-components are compared
++ (by an inclusion-test), or how the invertibility test is made in
++ the tower of simple extensions associated with a regular set.
++ These operations are realized respectively by the domain \spad{TS}
++ and the packages
++ \axiomType{QCMPACK}(R,E,V,TS) and \axiomType{RSETGCD}(R,E,V,P,TS).
++ The same way it does not care about the way univariate polynomial
++ gcd (with coefficients in the tower of simple extensions associated
++ with a regular set) are computed. The only requirement is that these
++ gcd need to have invertible initials (normalized or not).
++ WARNING. There is no need for a user to call directly any operation
++ of this package since they can be accessed by the domain \axiom{TS}.
++ Thus, the operations of this package are not documented.
RegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Wip ==> Record(done: Split, todo: List LpWT)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)
regsetgcdpack ==> RegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

KrullNumber: (LP, Split) -> N
numberOfVariables: (LP, Split) -> N
algebraicDecompose: (P,TS,B) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS,N,B) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
decompose: (LP, Split, B, B) -> Split
decompose: (LP, Split, B, B, B, B) -> Split
upDateBranches: (LP,Split,List LpWT,Wip,N) -> List LpWT
convert: Record(val: List P,tower: TS) -> String
printInfo: (List Record(val: List P,tower: TS), N) -> Void

Implementation == add

KrullNumber(lp: LP, lts: Split): N ==
ln: List N := [#(ts) for ts in lts]
n := #lp + reduce(max,ln)

numberOfVariables(lp: LP, lts: Split): N ==
lv: List V := variables([lp]$PS)
for ts in lts repeat lv := concat(variables(ts), lv)
# removeDuplicates(lv)
ground? p =>
   error " in algebraicDecompose$REGSET: should never happen !"

v := mvar(p); n := #ts
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
ts_v := select(ts,v)::P

if mdeg(p) < mdeg(ts_v)
   then
      lgwt := internalLastSubResultant(ts_v,p,ts_v_-,true,false)$regsetgcdpack
   else
      lgwt :=.internalLastSubResultant(p,ts_v,ts_v_-,true,false)$regsetgcdpack

lts: Split := []
llpwt: List LpWT := []

for gwt in lgwt repeat
   g := gwt.val; us := gwt.tower
   zero? g =>
      error " in algebraicDecompose$REGSET: should never happen !!"
   ground? g => "leave"
   if mvar(g) = v then lts := concat(augment(members(ts_v_+),augment(g,us)),lts)
   h := leadingCoefficient(g,v)
b: Boolean := purelyAlgebraic?(us)
lsfp := squareFreeFactors(h)$polsetpack
lus := augment(members(ts_v_+),augment(ts_v,us)@Split)
for f in lsfp repeat
   ground? f => "leave"
b and purelyAlgebraic?(f,us) => "leave"
   for vs in lus repeat
      llpwt := cons([[f,p],vs]$LpWT, llpwt)
[lts,llpwt]

lts: Split
if #ts < bound
   then
      lts := augment(p,ts)
   else
      lts := []
llpwt: List LpWT := []
[lts,llpwt]

lts: Split:= augment(p,ts)
llpwt: List LpWT := []
[lts,llpwt]

clos? => internalDecompose(p,ts,bound)
internalDecompose(p,ts)
-- ASSUME p not constant
llpwt: List LpWT := []
lts: Split := []
-- EITHER mvar(p) is null
if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
  then
    llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
    p := (p exquo lmp)::P
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
-- OR init(p) is null or not
lbwt := invertible?(ip,ts)@(List BWT)
for bwt in lbwt repeat
  bwt.val =>
    if algebraic?(mvar(p),bwt.tower)
      then
        rsl := algebraicDecompose(p,bwt.tower,true)
      else
        rsl := transcendentalDecompose(p,bwt.tower,bound)
    lts := concat(rsl.done,lts)
    llpwt := concat(rsl.todo,llpwt)
    -- purelyAlgebraicLeadingMonomial?(ip,bwt.tower) => "leave" -- UNPROVED CRITERIA
    purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave" -- SAFE
    (not ground? ip) =>
      zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
      (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
    riv := removeZero(ip,bwt.tower)
    (zero? riv) =>
      zero? tp => lts := cons(bwt.tower,lts)
      (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
    llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
  [lts,llpwt]

-- ASSUME p not constant
llpwt: List LpWT := []
lts: Split := []
-- EITHER mvar(p) is null
if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
  then
    llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
    p := (p exquo lmp)::P
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
-- OR init(p) is null or not
lbwt := invertible?(ip,ts)@(List BWT)
for bwt in lbwt repeat
  bwt.val =>
if algebraic?(mvar(p),bwt.tower)
then
    rsl := algebraicDecompose(p,bwt.tower,false)
else
    rsl := transcendentalDecompose(p,bwt.tower)
lts := concat(rsl.done,lts)
llpwt := concat(rsl.todo,llpwt)
purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave"
(not ground? ip) =>
    zero? tp => llpwt := cons([ip],bwt.tower)$LpWT, llpwt
(not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
riv := removeZero(ip,bwt.tower)
(zero? riv) =>
    zero? tp => lts := cons(bwt.tower,lts)
    (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
[lts,llpwt]
decompose(lp,lts,false,false,clos?,true,info?)
convert(lpwt: LpWT): String ==
ls: List String := ["<", string((#(lpwt.val))::Z), ",", string((#(lpwt.tower))::Z), ">"]
concat ls
printInfo(toSee: List LpWT, n: N): Void ==
lpwt := first toSee
s: String := concat ["[", string((#toSee)::Z), ",", convert(lpwt)$String]
m: N := #(lpwt.val)
toSee := rest toSee
for lpwt in toSee repeat
    m := m + #(lpwt.val)
    s := concat [s, " -> |", string(m::Z), "; {", string(n::Z),"}]
iprint(s)$iprintpack
void()
- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
- if clos? then SOLVE in the closure sense
- if rem? then REDUCE the current p by using remainder
- if info? then PRINT info
empty? lp => lts
branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$$quasicomppack
empty? branches => []
toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
toSave: Split := []
if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
while (not empty? toSee) repeat
if info? then printInfo(toSee,#toSave)
lpwt := first toSee; toSee := rest toSee
lp := lpwt.val; ts := lpwt.tower
empty? lp =>
    toSave := cons(ts, toSave)
p := first lp; lp := rest lp
if rem? and (not ground? p) and (not empty? ts)
    then
        p := remainder(p,ts).polnum
        p := removeZero(p,ts)
        zero? p => toSee := cons([lp,ts]$LpWT, toSee)
        ground? p => "leave"
        rsl := internalDecompose(p,ts,bound,clos?)
toSee := upDateBranches(lp,toSave,toSee,rsl,bound)
removeSuperfluousQuasiComponents(toSave)$quasicomppack

newBranches: List LpWT := wip.todo
newComponents: Split := wip.done
branches1, branches2: List LpWT
branches1 := []; branches2 := []
for branch in newBranches repeat
    us := branch.tower
    #us > n => "leave"
    newleq := sort(infRittWu?,concat(leq,branch.val))
    --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
    --any?(ground?,foo) => "leave"
    branches1 := cons([newleq,us]$LpWT, branches1)
for us in newComponents repeat
    #us > n => "leave"
    subQuasiComponent?(us,lts)$quasicomppack => "leave"
    --newleq := leq
    --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
    --any?(ground?,foo) => "leave"
    branches2 := cons([leq,us]$LpWT, branches2)
empty? branches1 =>
    empty? branches2 => current
    concat(branches2, current)
branches := concat([branches2, branches1, current])
-- branches := concat(branches,current)
removeSuperfluousCases(branches)$quasicomppack

——

— RSDCMPK.dotabb —

"RSDCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RSDCMPK"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
package RSETGCD RegularTriangularSetGcdPackage

-- RegularTriangularSetGcdPackage.input --

)set break resume
)sys rm -f RegularTriangularSetGcdPackage.output
)spool RegularTriangularSetGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RegularTriangularSetGcdPackage
--E 1

)spool
)lisp (bye)

-- RegularTriangularSetGcdPackage.help --

=================================================================================
RegularTriangularSetGcdPackage examples
=================================================================================

An internal package for computing gcds and resultants of univariate polynomials with coefficients in a tower of simple extensions of a field.

See Also:
  o )show RegularTriangularSetGcdPackage
RegularTriangularSetGcdPackage (RSETGCD)

Exports:
stopTableGcd!  stopTableInvSet!  integralLastSubResultant
prepareSubResAlgo  startTableGcd!  startTableInvSet!
toseInvertible?  toseInvertibleSet  toseLastSubResultant
toseSquareFreePart

--- package RSETGCD RegularTriangularSetGcdPackage ---

)abbrev package RSETGCD RegularTriangularSetGcdPackage
++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 08/30/1998
++ Date Last Updated: 12/15/1998
++ References :
++ [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++ algebraic towers of simple extensions" In proceedings of AAECC11
++ d'extensions simples et resolution des systemes d'équations
++ Description:
++ An internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.

RegularTriangularSetGcdPackage(R,E,V,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N => NonNegativeInteger
Z => Integer
B => Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union/Branch,"failed")
Split ==> List TS
KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
EntryGcd ==> List PWT
Hgcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
KeyInvSet ==> Record(arg1: P, arg3: TS)
EntryInvSet ==> List TS
HinvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)
Exports == with
   startTableGcd!: (S,S,S) -> Void
      ++ \axiom{startTableGcd!(s1,s2,s3)}
      ++ is an internal subroutine, exported only for developement.
   stopTableGcd!: () -> Void
      ++ \axiom{stopTableGcd!()} is an internal subroutine, exported only for developement.
   startTableInvSet!: (S,S,S) -> Void
      ++ \axiom{startTableInvSet!(s1,s2,s3)} is an internal subroutine, exported only for development.
   stopTableInvSet!: () -> Void
      ++ \axiom{stopTableInvSet!()} is an internal subroutine, exported only for development.
prepareSubResAlgo: (P,P,TS) -> List LpWT
   ++ \axiom{prepareSubResAlgo(p1,p2,ts)} is an internal subroutine, exported only for development.
   integralLastSubResultant: (P,P,TS) -> List PWT
      ++ \axiom{integralLastSubResultant(p1,p2,ts)} is an internal subroutine, exported only for development.
   toseLastSubResultant: (P,P,TS) -> List PWT
      ++ \axiom{toseLastSubResultant(p1,p2,ts)} has the same specifications as lastSubResultant from RegularTriangularSetCategory.
   toseInvertible?: (P,TS) -> B
      ++ \axiom{toseInvertible?(p1,p2,ts)} has the same specifications as invertible? from RegularTriangularSetCategory.
\textbf{toseInvertible?} \((p, ts) \rightarrow \text{List BWT}\)

++ \texttt{\textbackslash axiom\{toseInvertible?(p1,p2,ts)\}} has the same specifications as
++ \texttt{invertible?} from \texttt{RegularTriangularSetCategory}.

\textbf{toseInvertibleSet} \((p, ts) \rightarrow \text{Split}\)

++ \texttt{\textbackslash axiom\{toseInvertibleSet\{p1,p2,ts\}\}} has the same specifications as
++ \texttt{invertibleSet} from \texttt{RegularTriangularSetCategory}.

\textbf{toseSquareFreePart} \((p, ts) \rightarrow \text{List FWT}\)

++ \texttt{\textbackslash axiom\{toseSquareFreePart\{p,ts\}\}} has the same specifications as
++ \texttt{squareFreePart} from \texttt{RegularTriangularSetCategory}.

\textbf{Implementation == add}\n
\texttt{startTableGcd!\{ok: S, ko: S, domainName: S\}: Void ==}
\texttt{initTable!()$HGcd}
\texttt{printStats!(ok,ko)$HGcd}
\texttt{startStats!(domainName)$HGcd}
\texttt{void()}

\texttt{stopTableGcd!(): Void ==}
\texttt{if makingStats?()$HGcd then printStats!()$HGcd}
\texttt{clearTable!()$HGcd}

\texttt{startTableInvSet!\{ok: S, ko: S, domainName: S\}: Void ==}
\texttt{initTable!()$HInvSet}
\texttt{printStats!(ok,ko)$HInvSet}
\texttt{startStats!(domainName)$HInvSet}
\texttt{void()}

\texttt{stopTableInvSet!(): Void ==}
\texttt{if makingStats?()$HInvSet then printStats!()$HInvSet}
\texttt{clearTable!()$HInvSet}

\textbf{toseInvertible?\{p:P,ts:TS\}: Boolean ==}
\texttt{q := primitivePart initiallyReduce(p,ts)}
\texttt{zero? q => false}
\texttt{normalized?(q,ts) => true}
\texttt{v := mvar(q)}
\texttt{not algebraic?(v,ts) =>}
\texttt{toCheck: List BWT := toseInvertible?\{p,ts\}@\{List BWT\}}
\texttt{for bwt in toCheck repeat}
\texttt{bwt.val = false => return false}
\texttt{return true}

\texttt{ts_v := select(ts,v)::P}
\texttt{ts_v- := collectUnder\{ts,v\}}
\texttt{lgwt := internalLastSubResultant\{ts_v,q,ts_v-,false,\true\}}
\texttt{for gwt in lgwt repeat}
\texttt{g := gwt.val;}
\texttt{(not ground? g) and (mvar\{g\} = v) =>}
\texttt{return false}
\texttt{true}
  q := primitivePart initiallyReduce(p, ts)
  zero? q => [[false, ts]$BWT]
  normalized?(q, ts) => [[true, ts]$BWT]
  v := mvar(q)
  not algebraic?(v, ts) =>
    lbwt: List BWT := []
    toCheck: List BWT := toseInvertible?(init(q), ts)$@List BWT
    for bwt in toCheck repeat
      bwt.val => lbwt := cons(bwt, lbwt)
      newq := removeZero(q, bwt.tower)
      zero? newq => lbwt := cons(bwt, lbwt)
      lbwt := concat(toseInvertible?(newq, bwt.tower)$@List BWT, lbwt)
    return lbwt

  ts_v := select(ts, v)::P
  ts_v_- := collectUnder(ts, v)
  ts_v_+ := collectUpper(ts, v)
  lgwt := internalLastSubResultant(ts_v, q, ts_v_-, false, false)
  lbwt: List BWT := []
  for gwt in lgwt repeat
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
      ts := internalAugment(ts_v, ts)
      ts := internalAugment(members(ts_v_+), ts)
      lbwt := cons([true, ts]$BWT, lbwt)
    g := mainPrimitivePart g
    ts_g := internalAugment(g, ts)
    ts_g := internalAugment(members(ts_v_+), ts_g)
    -- USE internalAugment with parameters ?
    lbwt := cons([false, ts_g]$BWT, lbwt)
    h := lazyPquo(ts_v, g)
    (ground? h) or (mvar(h) < v) => "leave"
    h := mainPrimitivePart h
    ts_h := internalAugment(h, ts)
    ts_h := internalAugment(members(ts_v_+), ts_h)
    -- USE internalAugment with parameters ?
    -- CAN BE OPTIMIZED if the input tower is separable
    inv := toseInvertible?(q, ts_h)$@List BWT
    lbwt := concat([bwt for bwt in inv | bwt.val], lbwt)
  sort((x, y) +-> x.val < y.val, lbwt)

  k: KeyInvSet := [p, ts]
  e := extractIfCan(k)$HInvSet
  e case EntryInvSet => e::EntryInvSet
  q := primitivePart initiallyReduce(p, ts)
  zero? q => []
  normalized?(q, ts) => [ts]
  v := mvar(q)
toSave: Split := []
not algebraic?(v,ts) =>
toCheck: List BWT := toseInvertible?(init(q),ts)@(List BWT)
for bwt in toCheck repeat
  bwt.val => toSave := cons(bwt.tower,toSave)
  newq := removeZero(q,bwt.tower)
  zero? newq => "leave"
  toSave := concat(toseInvertibleSet(newq,bwt.tower), toSave)
  toSave := removeDuplicates toSave
return algebraicSort(toSave)$quasicomppack
ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := internalLastSubResultant(ts_v,q,ts_v_-,false,false)
for gwt in lgwt repeat
  g := gwt.val; ts := gwt.tower
  (ground? g) or (mvar(g) < v) =>
  ts := internalAugment(ts_v,ts)
  ts := internalAugment(members(ts_v_+),ts)
  toSave := cons(ts,toSave)
  g := mainPrimitivePart g
  h := lazyPquo(ts_v,g)
  h := mainPrimitivePart h
  (ground? h) or (mvar(h) < v) => "leave"
  ts_h := internalAugment(h,ts)
  ts_h := internalAugment(members(ts_v_+),ts_h)
  inv := toseInvertibleSet(q,ts_h)
  toSave := removeDuplicates concat(inv,toSave)
toSave := algebraicSort(toSave)$quasicomppack
insert!(k,toSave)$HInvSet
toSave

-- ASSUME p is not constant and mvar(p) > mvar(ts)
-- ASSUME init(p) is invertible w.r.t. ts
-- ASSUME p is mainly primitive
-- one? mdeg(p) => [[p,ts]$PWT]
-- mdeg(p) = 1 => [[p,ts]$PWT]
v := mvar(p)$P
q: P := mainPrimitivePart D(p,v)
lgvt: List PWT := internalLastSubResultant(p,q,ts,true,false)
lpwt := List PWT := []
sfp : P
for gwt in lgvt repeat
  g := gwt.val; us := gwt.tower
  (ground? g) or (mvar(g) < v) =>
  lpwt := cons([p,us],lpwt)
  g := mainPrimitivePart g
  sfp := lazyPquo(p,g)
sfp := mainPrimitivePart stronglyReduce(sfp,us)
lpwt := cons([sfp,us],lpwt)
lpwt
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME init(p1) invertible modulo ts !!!
toSee: List LpWT := [[[p1,p2],ts]$LpWT]
toSave: List LpWT := []
v := mvar(p1)
while (not empty? toSee) repeat
  lpwt := first toSee; toSee := rest toSee
  p1 := lpwt.val.1; p2 := lpwt.val.2
  ts := lpwt.tower
  lbwt := toseeInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
  for bwt in lbwt repeat
    (bwt.val = true) and (degree(p2,v) > 0) =>
      p3 := prem(p1, -p2)
      s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
      toSave := cons([[p2,p3,s],bwt.tower]$LpWT,toSave)
      -- p2 := initiallyReduce(p2,bwt.tower)
      newp2 := primitivePart initiallyReduce(p2,bwt.tower)
      (bwt.val = true) =>
        -- toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
        toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
        -- zero? newp2 =>
        toSave := cons([[p1,0,1],bwt.tower]$LpWT,toSave)
        -- toSee := cons([[p1,newp2],bwt.tower]$LpWT,toSee)
        toSee := cons([[p1,newp2],bwt.tower]$LpWT,toSee)
  toSave
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME p1 and p2 have no algebraic coefficients
  lsr := lastSubResultant(p1, p2)
  ground?(lsr) => [[lsr,ts]$PWT]
  mvar(lsr) < mvar(p1) => [[lsr,ts]$PWT]
  gili2i := gcd(init(p1),init(p2))
  ex: Union(P,"failed") := (gili2i * lsr) exquo$P init(lsr)
  ex case "failed" => [[lsr,ts]$PWT]
  [[ex::P,ts]$PWT]
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- if b1 ASSUME init(p2) invertible w.r.t. ts
  -- if b2 BREAK with the first non-trivial gcd
k: KeyGcd := [p1,p2,ts,b2]
e ::= extractIfCan(k)$HGcd
e case EntryGcd => e::EntryGcd
toSave: List PWT
empty? ts =>
toSave := integralLastSubResultant(p1,p2,ts)
insert!(k,toSave)$HGcd
return toSave
toSee: List LpWT
if b1 then
  p3 := prem(p1, -p2)
s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
toSee := [[[p2,p3,s],ts]$LpWT]
else
toSee := prepareSubResAlgo(p1,p2,ts)
toSave := internalLastSubResultant(toSee,mvar(p1),b2)
insert!(k,toSave)$HGcd
return toSave

internalLastSubResultant(llpwt: List LpWT,v:V,b2:B): List PWT ==
toReturn: List PWT := []; toSee: List LpWT;
while (not empty? llpwt) repeat
  toSee := llpwt; llpwt := []
  -- CONSIDER FIRST the vanishing current last subresultant
  for lpwt in toSee repeat
    p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3; ts := lpwt.tower
    lbwt := toseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
    for bwt in lbwt repeat
      bwt.val = false =>
      toReturn := cons([p1,bwt.tower]$PWT, toReturn)
      b2 and positive?(degree(p1,v)) => return toReturn
      llpwt := cons([[p1,p2,s],bwt.tower]$LpWT, llpwt)
  empty? llpwt => "leave"
  -- CONSIDER NOW the branches where the computations continue
  toSee := llpwt; llpwt := []
lpwt := first toSee; toSee := rest toSee
  p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
  delta: N := (mdeg(p1) - degree(p2,v))::N
  p3: P := LazardQuotient2(p2, leadingCoefficient(p2,v), s, delta)
  zero?(degree(p3,v)) =>
  toReturn := cons([[p3,lpwt.tower]$PWT, toReturn)
  for lpwt in toSee repeat
    toReturn := cons([[p3,lpwt.tower]$PWT, toReturn)
    (p1, p2) := (p3, next_subResultant2(p1, p2, p3, s))
    s := leadingCoefficient(p1,v)
lpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
  for lpwt in toSee repeat
    llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
toReturn
ground? p1 =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #1"
ground? p2 =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
not (mvar(p2) = mvar(p1)) =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
algebraic?(mvar(p1),ts) =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #1"
not initiallyReduced?(p1,ts) =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #1"
not initiallyReduced?(p2,ts) =>
   error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
purelyTranscendental?(p1,ts) and purelyTranscendental?(p2,ts) =>
   integralLastSubResultant(p1,p2,ts)
if mdeg(p1) < mdeg(p2) then
   (p1, p2) := (p2, p1)
   if odd?(mdeg(p1)) and odd?(mdeg(p2)) then p2 := - p2
   internalLastSubResultant(p1,p2,ts,false,false)

package REPDB RepeatedDoubling

— RepeatedDoubling.input —

)set break resume
)sys rm -f RepeatedDoubling.output
)spool RepeatedDoubling.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show RepeatedDoubling
--E 1

)spool
)lisp (bye)

---

— RepeatedDoubling.help —

====================================================================
RepeatedDoubling examples
====================================================================

Implements multiplication by repeated addition

See Also:
o )show RepeatedDoubling

---

RepeatedDoubling (REPDB)

Exports:
double

— package REPDB RepeatedDoubling —

)abbrev package REPDB RepeatedDoubling
++ Description:
++ Implements multiplication by repeated addition

-- the following package is only instantiated over %
thus shouldn’t be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedDoubling, $mutableDomains)

RepeatedDoubling(S):Exports ==Implementation where
S: SetCategory with
"+":(%,%)->%
++ x+y returns the sum of x and y
Exports == with
double: (PositiveInteger,S) -> S
++ double(i, r) multiplies r by i using repeated doubling.
Implementation == add
x: S
n: PositiveInteger
double(n,x) ==
-- one? n => x
(n = 1) => x
odd?(n)$Integer =>
x + double(shift(n,-1) pretend PositiveInteger,(x+x))
double(shift(n,-1) pretend PositiveInteger,(x+x))

package REPSQ RepeatedSquaring

— RepeatedSquaring.input —
This code computes $x^n$ by repeated squaring. It takes advantage of the law of exponents so that if $n = a \times b$ then

$$x^n = x^a \times x^b$$

It also takes advantage of the fact that $n$ has a binary representation so that a one-bit right-shift of a number divides it by 2. Thus we get three cases:

- $n = 1 \rightarrow x^1$
- $n$ odd $\rightarrow$ repeatedsquare($x^x, n/2$)$\times x$
- $n$ even $\rightarrow$ repeatedsquare($x^x, n/2$)

Since each recursive call divides $n$ by 2 the algorithm is $O(\log(n))$.

See Also:
- )show RepeatedSquaring
RepeatedSquaring (REPSQ)

Exports:
  expt

— package REPSQ RepeatedSquaring —

)abbrev package REPSQ RepeatedSquaring
++ Description:
++ Implements exponentiation by repeated squaring

-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedSquaring, $mutableDomains)

RepeatedSquaring(S):Exports == Implementation where
  S: SetCategory with
    "*":(%,%)->%
      ++ x*y returns the product of x and y
  Exports == with
    expt: (S,PositiveInteger) -> S
      ++ expt(r, i) computes r**i by repeated squaring
  Implementation == add
    x: S
    n: PositiveInteger
    expt(x, n) ==
      -- one? n => x
      (n = 1) => x
      odd?(n)$Integer=> x * expt(x*x,shift(n,-1) pretend PositiveInteger)
      expt(x*x,shift(n,-1) pretend PositiveInteger)
package REP1 RepresentationPackage1

--- RepresentationPackage1.input ---

)set break resume
)sys rm -f RepresentationPackage1.output
)spool RepresentationPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RepresentationPackage1
--E 1

)spool
)lisp (bye)

--- RepresentationPackage1.help ---

====================================================================
RepresentationPackage1 examples
====================================================================

RepresentationPackage1 provides functions for representation theory for finite groups and algebras.

The package creates permutation representations and uses tensor products and its symmetric and antisymmetric components to create new representations of larger degree from given ones.

Note that instead of having parameters from Permutation this package
allows list notation of permutations as well: e.g. \([1,4,3,2]\) denotes
permutes 2 and 4 and fixes 1 and 3.

See Also:
\(\)show RepresentationPackage1

RepresentationPackage1 (REP1)

Exports:
antisymmetricTensors createGenericMatrix permutationRepresentation
symmetricTensors tensorProduct

— package REP1 RepresentationPackage1 —

)abbrev package REP1 RepresentationPackage1
++ Authors: Holger Gollan, Johannes Grabmeier, Thorsten Werther
++ Date Created: 12 September 1987
++ Date Last Updated: 24 May 1991
++ References:
++ G. James, A. Kerber: The Representation Theory of the Symmetric
++ J. Grabmeier, A. Kerber: The Evaluation of Irreducible
++ Polynomial Representations of the General Linear Groups
++ and of the Unitary Groups over Fields of Characteristic 0,
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
++ Description:
++ RepresentationPackage1 provides functions for representation theory
++ for finite groups and algebras.
++ The package creates permutation representations and uses tensor products
++ and its symmetric and antisymmetric components to create new
++ representations of larger degree from given ones.
++ Note that instead of having parameters from \spadtype{Permutation}
++ this package allows list notation of permutations as well:
++ e.g. \spad{[1,4,3,2]} denotes permutes 2 and 4 and fixes 1 and 3.

RepresentationPackage1(R): public == private where

R : Ring  OF ==> OutputForm
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
I ==> Integer
L ==> List
M ==> Matrix
P ==> Polynomial
SM ==> SquareMatrix
V ==> Vector
ICF ==> IntegerCombinatoricFunctions Integer
SGCF ==> SymmetricGroupCombinatoricFunctions
PERM ==> Permutation

public ==> with

if R has commutative("*") then
  antisymmetricTensors : (M R,PI) -> M R
  ++ antisymmetricTensors(a,n) applies to the square matrix
  ++ a the irreducible, polynomial representation of the
  ++ general linear group GL_m, where m is the number of
  ++ rows of a, which corresponds to the partition
  ++ \((1,1,\ldots,1,0,0,\ldots,0)\) of n.
  ++ Error: if n is greater than m.
  ++ Note that this corresponds to the symmetrization of the
  ++ representation with the sign representation of the symmetric group
  ++ Sn. The carrier spaces of the representation are the antisymmetric
  ++ tensors of the n-fold tensor product.
  if R has commutative("*") then
    antisymmetricTensors : (L M R, PI) -> L M R
    ++ antisymmetricTensors(la,n) applies to each
    ++ m-by-m square matrix in
    ++ the list la the irreducible, polynomial representation
    ++ of the general linear group GL_m
    ++ which corresponds
    ++ to the partition \((1,1,\ldots,1,0,0,\ldots,0)\) of n.
    ++ Error: if n is greater than m.
    ++ Note that this corresponds to the symmetrization of the
    ++ representation with the sign representation of the symmetric group
    ++ Sn. The carrier spaces of the representation are the antisymmetric
    ++ tensors of the n-fold tensor product.
createGenericMatrix : NNI -> M P R
++ createGenericMatrix(m) creates a square matrix of dimension k
++ whose entry at the i-th row and j-th column is the
++ indeterminate x[i,j] (double subscripted).

++ symmetricTensors : (M R, PI) -> M R
++ symmetricTensors(a,n) applies to the m-by-m
++ square matrix a the
++ irreducible, polynomial representation of the general linear
++ group GLm
++ which corresponds to the partition (n,0,...,0) of n.
++ Error: if a is not a square matrix.
++ Note that this corresponds to the symmetrization of the
++ representation with the trivial representation of the symmetric
++ group Sn.
++ The carrier spaces of the representation are the symmetric
++ tensors of the n-fold tensor product.

++ symmetricTensors : (L M R, PI) -> L M R
++ symmetricTensors(la,n) applies to each m-by-m square matrix in the
++ list la the irreducible, polynomial representation
++ of the general linear group GLm
++ which corresponds
++ to the partition (n,0,...,0) of n.
++ Error: if the matrices in la are not square matrices.
++ Note that this corresponds to the symmetrization of the
++ representation with the trivial representation of the symmetric
++ group Sn.
++ The carrier spaces of the representation are the symmetric
++ tensors of the n-fold tensor product.

++ tensorProduct : (M R, M R) -> M R
++ tensorProduct(a,b) calculates the Kronecker product
++ of the matrices a and b.
++ Note that if each matrix corresponds to a group representation
++ (repr. of generators) of one group, then these matrices
++ correspond to the tensor product of the two representations.

++ tensorProduct : (L M R, L M R) -> L M R
++ tensorProduct([a1,...,ak],[b1,...,bk]) calculates the list of
++ Kronecker products of the matrices ai and bi
++ for {1 <= i <= k}.
++ Note that if each list of matrices corresponds to a group
++ representation (repr. of generators) of one group, then these
++ matrices correspond to the tensor product of the two representations.

++ tensorProduct : M R -> M R
++ tensorProduct(a) calculates the Kronecker product
++ of the matrix a with itself.

++ tensorProduct : L M R -> L M R
++ tensorProduct([a1,...,ak]) calculates the list of
++ Kronecker products of each matrix ai with itself
++ for {1 <= i <= k}.
++ Note that if the list of matrices corresponds to a group
++ representation (repr. of generators) of one group, then these
++ matrices correspond to the tensor product of the representation.
++ with itself.

permutationRepresentation : (PERM I, I) -> M I
++ permutationRepresentation(pi,n) returns the matrix
++ (deltai,pi(i)) (Kronecker delta) for a permutation
++ pi of \{1,2,...,n\}.

permutationRepresentation : L I -> M I
++ permutationRepresentation(pi,n) returns the matrix
++ (deltai,pi(i)) (Kronecker delta) if the permutation
++ pi is in list notation and permutes \{1,2,...,n\}.

permutationRepresentation : (L PERM I, I) -> L M I
++ permutationRepresentation([pi1,...,pik],n) returns the list
++ of matrices [(deltai,pi1(i)),...,deltai(pik(i))]
++ (Kronecker delta) for the permutations pi1,...,pik
++ of \{1,2,...,n\}.

permutationRepresentation : L L I -> L M I
++ permutationRepresentation([pi1,...,pik],n) returns the list
++ of matrices [(deltai,pi1(i)),...,deltai(pik(i))]
++ if the permutations pi1,...,pik are in
++ list notation and are permuting \{1,2,...,n\}.

private ==> add

-- import of domains and packages

import OutputForm

-- declaration of local functions:

calcCoef : (L I, M I) -> I
-- calcCoef(beta,C) calculates the term
-- |S(beta) gamma S(alpha)| / |S(beta)|

invContent : L I -> V I
-- invContent(alpha) calculates the weak monoton function f with
-- f : m -> n with invContent alpha. f is stored in the returned
-- vector

-- definition of local functions

calcCoef(beta,C) ==
prod : I := 1
for i in 1..maxIndex beta repeat
  prod := prod * multinomial(beta(i), entries row(C,i))$ICF
prod
invContent(alpha) ==
  n : NNI := (+/alpha)::NNI
  f : V I := new(n,0)
  i : NNI := 1
  j : I := -1
  for og in alpha repeat
    j := j + 1
    for k in 1..og repeat
      f(i) := j
      i := i + 1
  f

-- exported functions:

if R has commutative("*") then
  antisymmetricTensors ( a : M R , k : PI ) ==
    n : NNI := nrows a
    m : I := binomial(n,k)$ICF
    il : L L I := [subSet(n,k,i)$SGCF for i in 0..m-1]
    b : M R := zero(m::NNI, m::NNI)
    for i in 1..m repeat
      c : M R := zero(k,k)
      lr : L I := il.i
      lt : L I := il.j
      for r in 1..k repeat
        for t in 1..k repeat
          rr : I := lr.r
          tt : I := lt.t
          -c.r.t := a.(1+rr).(1+tt)
          setelt(c,r,t,elt(a, 1+rr, 1+tt))
      setelt(b, i, j, determinant c)
    b

if R has commutative("*") then
  antisymmetricTensors(la: L M R, k: PI) ==
    [antisymmetricTensors(ma,k) for ma in la]

symmetricTensors (a : M R, n : PI) ==
m : NNI := nrows a
m ^= ncols a =>
  error("Input to symmetricTensors is no square matrix")

n := 1 => a

dim : NNI := (binomial(m+n-1,n)$ICF)::NNI

c : M R := new(dim,dim,0)
f : V I := new(n,0)
g : V I := new(n,0)
nullMatrix : M I := new(1,1,0)
colemanMatrix : M I

for i in 1..dim repeat
  -- unrankImproperPartitions1 starts counting from 0
  alpha := unrankImproperPartitions1(n,m,i-1)$SGCF
  f := invContent(alpha)
  for j in 1..dim repeat
    -- unrankImproperPartitions1 starts counting from 0
    beta := unrankImproperPartitions1(n,m,j-1)$SGCF
    g := invContent(beta)
    colemanMatrix := nextColeman(alpha,beta,nullMatrix)$SGCF
    while colemanMatrix ^= nullMatrix repeat
      gamma := inverseColeman(alpha,beta,colemanMatrix)$SGCF
      help := calcCoef(beta,colemanMatrix)::R
      for k in 1..n repeat
        -- end of while
        -- end of j-loop
        -- end of i-loop
  c

symmetricTensors(la : L M R, k : PI) ==
  [symmetricTensors (ma, k) for ma in la]

tensorProduct(a: M R, b: M R) ==
  n := NNI := nrows a
  m := NNI := nrows b
  nc := NNI := ncols a
  mc := NNI := ncols b
  c := M R := zero(n * m, nc * mc)
indexr := NNI := 1 -- row index
for i in 1..n repeat
  for k in 1..m repeat
    indexc := NNI := 1 -- column index
    for j in 1..nc repeat
for l in 1..mc repeat
  c(indexr,indexc) := a(i,j) * b(k,l)
  indexc := indexc + 1
  indexr := indexr + 1
  c

tensorProduct (la: L M R, lb: L M R) ==
  [tensorProduct(la.i, lb.i) for i in 1..maxIndex la]

tensorProduct(a : M R) == tensorProduct(a, a)

tensorProduct(la : L M R) ==
  tensorProduct(la :: L M R, la :: L M R)

permutationRepresentation (p : PERM I, n : I) ==
  -- permutations are assumed to permute {1,2,...,n}
  a : M I := zero(n :: NNI, n :: NNI)
  for i in 1..n repeat
    a(eval(p,i)$PERM I),i) := 1
  a

permutationRepresentation (p : L I) ==
  -- permutations are assumed to permute {1,2,...,n}
  n : I := #p
  a : M I := zero(n::NNI, n::NNI)
  for i in 1..n repeat
    a(p.i,i) := 1
  a

permutationRepresentation(listperm : L PERM I, n : I) ==
  -- permutations are assumed to permute {1,2,...,n}
  [permutationRepresentation(perm, n) for perm in listperm]

permutationRepresentation (listperm : L L I) ==
  -- permutations are assumed to permute {1,2,...,n}
  [permutationRepresentation perm for perm in listperm]

createGenericMatrix(m) ==
  res : M P R := new(m,m,0$(P R))
  for i in 1..m repeat
    for j in 1..m repeat
      iof : OF := coerce(i)$Integer
      jof : OF := coerce(j)$Integer
      le : L OF := cons(iof,list jof)
      sy : Symbol := subscript(x::Symbol, le)$Symbol
      res(i,j) := (sy :: P R)
package REP2 RepresentationPackage2

-- RepresentationPackage2.input --

)set break resume
/sys rm -f RepresentationPackage2.output
/spool RepresentationPackage2.output
/set message test on
/set message auto off
/clear all

--S 1 of 1
/show RepresentationPackage2
--E 1

/spool
/lisp (bye)

-- RepresentationPackage2.help --

====================================================================
RepresentationPackage2 examples
====================================================================

RepresentationPackage2 provides functions for working with
modular representations of finite groups and algebra.
The routines in this package are created, using ideas of R. Parker,
(the meat-Axe) to get smaller representations from bigger ones,
i.e. finding sub- and factormodules, or to show, that such the
representations are irreducible.

Note that most functions are randomized functions of Las Vegas type
i.e. every answer is correct, but with small probability
the algorithm fails to get an answer.

See Also:
  o )show RepresentationPackage2

RepresentationPackage2 (REP2)

Exports:
  areEquivalent?    completeEchelonBasis
  createRandomElement cyclicSubmodule
  isAbsolutelyIrreducible? meatAxe
  scanOneDimSubspaces split
  standardBasisOfCyclicSubmodule

— package REP2 RepresentationPackage2 —

)abbrev package REP2 RepresentationPackage2
++ Authors: Holger Gollan, Johannes Grabmeier
++ Date Created: 10 September 1987
++ Date Last Updated: 20 August 1990
++ Reference:
  ++ R. A. Parker: The Computer Calculation of Modular Characters
  ++ (The Meat-Axe), in M. D. Atkinson (Ed.), Computational Group Theory
  ++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
  ++ their Realization in the Computer Algebra System Scratchpad,
++ Description:
++ RepresentationPackage2 provides functions for working with
++ modular representations of finite groups and algebra.
++ The routines in this package are created, using ideas of R. Parker,
++ (the meat-Axe) to get smaller representations from bigger ones,
++ i.e. finding sub- and factormodules, or to show, that such the
++ representations are irreducible.
++ Note that most functions are randomized functions of Las Vegas type
++ i.e. every answer is correct, but with small probability
++ the algorithm fails to get an answer.

RepresentationPackage2(R): public == private where

R : Ring
OF ==> OutputForm
I ==> Integer
L ==> List
SM ==> SquareMatrix
M ==> Matrix
NNI ==> NonNegativeInteger
V ==> Vector
PI ==> PositiveInteger
B ==> Boolean
RADIX ==> RadixExpansion

public == with

completeEchelonBasis : V V R -> M R
++ completeEchelonBasis(lv) completes the basis lv assumed
++ to be in echelon form of a subspace of R**n (n the length
++ of all the vectors in lv with unit vectors to a basis of
++ R**n. It is assumed that the argument is not an empty
++ vector and that it is not the basis of the 0-subspace.
++ Note that the rows of the result correspond to the vectors
++ of the basis.
createRandomElement : (L M R, M R) -> M R
++ createRandomElement(aG,x) creates a random element of the group
++ algebra generated by aG.
randomWord : (L L I, L M) -> M R
--++ You can create your own 'random' matrix with "randomWord(lli, lm)".
--++ Each li in lli determines a product of matrices, the entries in li
--++ determine which matrix from lm is chosen. Finally we sum over all
--++ products. The result "sm" can be used to call split with (e.g.)
--++ second parameter "first nullSpace sm"
if R has EuclideanDomain then -- using rowEchelon
  cyclicSubmodule : (L M R, V R) -> V V R
  ++ cyclicSubmodule(lm,v) generates a basis as follows.
  ++ It is assumed that the size n of the vector equals the number
  ++ of rows and columns of the matrices. Then the matrices generate
  ++ a subalgebra, say \spad{A}, of the algebra of all square matrices of
  ++ dimension n. V R is an \spad{A}-module in the natural way.
++ cyclicSubmodule(lm,v) generates the $R$-Basis of $Av$ as
++ described in section 6 of R. A. Parker's "The Meat-Axe".
++ Note that in contrast to the description in "The Meat-Axe" and to
++ standardBasisOfCyclicSubmodule the result is in echelon form.
++ It is assumed that the size $n$ of the vector equals the number
++ of rows and columns of the matrices. Then the matrices generate
++ a subalgebra, say $\text{spad}(A)$,
++ of the algebra of all square matrices of
++ dimension $n$. $V_R$ is an $\text{spad}(A)$-module in the natural way.
++ standardBasisOfCyclicSubmodule(lm,v) calculates a matrix whose
++ non-zero column vectors are the $R$-Basis of $Av$ achieved
++ in the way as described in section 6
++ of R. A. Parker's "The Meat-Axe".
++ Note that in contrast to cyclicSubmodule, the result is not
++ in echelon form.

if $R$ has Field then -- only because of inverse in SM
++ cyclicSubmodule(lm,v) generates the $R$-Basis of $Av$ as
++ described in section 6 of R. A. Parker's "The Meat-Axe".
++ Note that in contrast to the description in "The Meat-Axe" and to
++ standardBasisOfCyclicSubmodule the result is in echelon form.
++ It is assumed that the size $n$ of the vector equals the number
++ of rows and columns of the matrices. Then the matrices generate
++ a subalgebra, say $\text{spad}(A)$,
++ of the algebra of all square matrices of
++ dimension $n$. $V_R$ is an $\text{spad}(A)$-module in the natural way.
++ standardBasisOfCyclicSubmodule(lm,v) calculates a matrix whose
++ non-zero column vectors are the $R$-Basis of $Av$ achieved
++ in the way as described in section 6
++ of R. A. Parker's "The Meat-Axe".
++ Note that in contrast to cyclicSubmodule, the result is not
++ in echelon form.

areEquivalent? : (L M R, L M R, B, I) -> M R
++ areEquivalent?(aG0,aG1,randomelements,numberOfTries) tests
++ whether the two lists of matrices, all assumed of same
++ square shape, can be simultaneously conjugated by a non-singular
++ matrix. If these matrices represent the same group generators,
++ the representations are equivalent.
++ The algorithm tries
++ numberOfTries times to create elements in the
++ generated algebras in the same fashion. If their ranks differ,
++ they are not equivalent. If an
++ isomorphism is assumed, then
++ the kernel of an element of the first algebra
++ is mapped to the kernel of the corresponding element in the
++ second algebra. Now consider the one-dimensional ones.
++ If they generate the whole space (e.g. irreducibility !)
++ we use standardBasisOfCyclicSubmodule to create the
++ only possible transition matrix. The method checks whether the
++ matrix conjugates all corresponding matrices from aG1.
++ The way to choose the singular matrices is as in meatAxe.
++ If the two representations are equivalent, this routine
++ returns the transformation matrix TM with
++ aG0.ij * TM = TM * aG1.ij for all i. If the representations
++ are not equivalent, a small 0-matrix is returned.
++ Note that the case
++ with different sets of group generators cannot be handled.
++ areEquivalent? : (L M R, L M R) -> M R
++ areEquivalent?(aG0,aG1) calls areEquivalent?(aG0,aG1,true,25).
++ Note that the choice of 25 was rather arbitrary.
areEquivalent? : (L M R, L M R, I) -> M R
++ areEquivalent?(aG0,aG1,numberOfTries) calls
++ areEquivalent?(aG0,aG1,true,25).
++ Note that the choice of 25 was rather arbitrary.
isAbsolutelyIrreducible? : (L M R, I) -> B
++ isAbsolutelyIrreducible?(aG, numberOfTries) uses
++ Norton's irreducibility test to check for absolute
++ irreducibility, assuming if a one-dimensional kernel is found.
++ As no field extension changes create "new" elements
++ in a one-dimensional space, the criterium stays true
++ for every extension. The method looks for one-dimensionals only
++ by creating random elements (no fingerprints) since
++ a run of meatAxe would have proved absolute irreducibility
++ anyway.

isAbsolutelyIrreducible? : L M R -> B
++ isAbsolutelyIrreducible?(aG) calls
++ isAbsolutelyIrreducible?(aG,25).
++ Note that the choice of 25 was rather arbitrary.

split : (L M R, V R) -> L L M R
++ split(aG, vector) returns a subalgebra \spad{A} of all
++ square matrix of dimension n as a list of list of matrices,
++ generated by the list of matrices aG, where n denotes both
++ the size of vector as well as the dimension of each of the
++ square matrices.
++ V R is an A-module in the natural way.
++ split(aG, vector) then checks whether the cyclic submodule
++ generated by vector is a proper submodule of V R.
++ If successful, it returns a two-element list, which contains
++ first the list of the representations of the submodule,
++ then the list of the representations of the factor module.
++ If the vector generates the whole module, a one-element list
++ of the old representation is given.
++ Note that a later version this should call the other split.

split: (L M R, V V R) -> L L M R
++ split(aG, submodule) uses a proper submodule of R**n
++ to create the representations of the submodule and of
++ the factor module.

if (R has Finite) and (R has Field) then

meatAxe : (L M R, B, I, I) -> L L M R
++ meatAxe(aG,randomElements,numberOfTries, maxTests) returns
++ a 2-list of representations as follows.
++ All matrices of argument aG are assumed to be square
++ and of equal size.
++ Then \spad{aG} generates a subalgebra, say \spad{A}, of the algebra
++ of all square matrices of dimension n. V R is an A-module
++ in the usual way.
++ meatAxe(aG,numberOfTries, maxTests) creates at most
++ numberOfTries random elements of the algebra, tests
++ then for singularity. If singular, it tries at most maxTests
++ elements of its kernel to generate a proper submodule.
++ If successful, a 2-list is returned: first, a list
++ containing first the list of the
++ representations of the submodule, then a list of the
++ representations of the factor module.
Otherwise, if we know that all the kernel is already scanned, Norton’s irreducibility test can be used either to prove irreducibility or to find the splitting. If randomElements is false, the first 6 tries use Parker’s fingerprints.

meatAxe : L M R -> L L M R
++ meatAxe(aG) calls meatAxe(aG,false,25,7) returns a 2-list of representations as follows.
++ All matrices of argument \spad{aG} are assumed to be square and of equal size. Then \spad{aG} generates a subalgebra, say \spad{A}, of the algebra of all square matrices of dimension n. \spad{V R} is an \spad{A}-module in the usual way.
++ meatAxe(aG) creates at most 25 random elements of the algebra, tests them for singularity. If singular, it tries at most 7 elements of its kernel to generate a proper submodule. If successful a list which contains first the list of the representations of the submodule, then a list of the representations of the factor module is returned.
++ Otherwise, if we know that all the kernel is already scanned, Norton’s irreducibility test can be used either to prove irreducibility or to find the splitting.
++ Notes: the first 6 tries use Parker’s fingerprints. Also, 7 covers the case of three-dimensional kernels over the field with 2 elements.

meatAxe: (L M R, B) -> L L M R
++ meatAxe(aG, randomElements) calls meatAxe(aG,false,6,7), only using Parker’s fingerprints, if randomElements is false. If it is true, it calls meatAxe(aG,true,25,7), only using random elements.
++ Note that the choice of 25 was rather arbitrary. Also, 7 covers the case of three-dimensional kernels over the field with 2 elements.

meatAxe : (L M R, PI) -> L L M R
++ meatAxe(aG, numberOfTries) calls meatAxe(aG,true,numberOfTries,7).
++ Notes: 7 covers the case of three-dimensional kernels over the field with 2 elements.

scanOneDimSubspaces: (L V R, I) -> V R
++ scanOneDimSubspaces(basis,n) gives a canonical representative of the n-th one-dimensional subspace of the vector space generated by the elements of basis, all from \spad{R**n}. The coefficients of the representative are of shape (0,...,0,1,*,...,*) in \spad{R}. If the size of \spad{R} is \spad{q}, then there are \spad{(q**n-1)/(q-1)} of them.
++ We first reduce n modulo this number, then find the largest i such that +/[q**i for i in 0..i-1] <= n.
++ Subtracting this sum of powers from n results in an
++ i-digit number to basis q. This fills the positions of the
++ stars.
-- would prefer to have (V V R,...) but nullSpace results
-- in L V R

private ==> add

-- import of domain and packages
import OutputForm

-- declarations and definitions of local variables and
-- local function

blockMultiply: (M R, M R, L I, I) -> M R
-- blockMultiply(a,b,li,n) assumes that a has n columns
-- and b has n rows, li is a sublist of the rows of a and
-- a sublist of the columns of b. The result is the
-- multiplication of the (li x n) part of a with the
-- (n x li) part of b. We need this, because just matrix
-- multiplying the parts would require extra storage.
blockMultiply(a, b, li, n) ==
    matrix([[ a[i,s] * b(s,j) for s in 1..n ]
        for j in li]
        for i in li)

fingerPrint: (NNI, M R, M R, M R) -> M R
-- is local, because one should know all the results for smaller i
fingerPrint (i : NNI, a : M R, b : M R, x : M R) ==
-- i > 2 only gives the correct result if the value of x from
-- the parameter list equals the result of fingerprint(i-1,...)
(i::PI) = 1 => x := a + b + a*b
(i::PI) = 2 => x := (x + a*b)*b
(i::PI) = 3 => x := a + b*x
(i::PI) = 4 => x := x + b
(i::PI) = 5 => x := x + a*b
(i::PI) = 6 => x := x - a + b*a
error "Sorry, but there are only 6 fingerprints!"
    x

-- definition of exported functions

randomWord(lli,lm) ==
-- we assume that all matrices are square of same size
-- numberOfMatrices := #lm
-- +/[l[m.(i+i rem numberOfMatrices) for i in li] for li in lli]

completeEchelonBasis(basis) ==

dimensionOfSubmodule : NNI := #basis
n : NNI := \# basis,1
indexOfVectorToBeScanned : NNI := 1
row : NNI := dimensionOfSubmodule

completedBasis : M R := zero(n, n)
for i in 1..dimensionOfSubmodule repeat
    completedBasis := setRow_!(completedBasis, i, basis.i)
if #basis <= n then
    newStart : NNI := 1
    for j in 1..n
        while indexOfVectorToBeScanned <= dimensionOfSubmodule repeat
            if basis.indexOfVectorToBeScanned.j = 0 then
                completedBasis(1+row,j) := 1 -- put unit vector into basis
                row := row + 1
            else
                indexOfVectorToBeScanned := indexOfVectorToBeScanned + 1
                newStart : NNI := j + 1
            end if
        end for
    end for
end if
completedBasis

createRandomElement(aG,algElt) ==
    numberOfGenerators : NNI := \#aG
    randomIndex := randnum numberOfGenerators
    algElt := algElt * aG.randomIndex
    randomIndex := randnum numberOfGenerators
    algElt + aG.randomIndex

if R has EuclideanDomain then
    cyclicSubmodule (lm : L M R, v : V R) ==
        basis : M R := rowEchelon matrix list entries v
        -- normalizing the vector
        -- all these elements lie in the submodule generated by v
        furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
        -- furtherElts has elements of the generated submodule. It will
        -- will be checked whether they are in the span of the vectors
        -- computed so far. Of course we stop if we have got the whole
        -- space.
        while (^null furtherElts) and (nrows basis < \#v) repeat
            w : V R := first furtherElts
            nextVector : M R := matrix list entries w -- normalizing the vector
            -- will the rank change if we add this nextVector
            -- to the basis so far computed?
            addedToBasis : M R := vertConcat(basis, nextVector)
            if rank addedToBasis ^= nrows basis then
                basis := rowEchelon addedToBasis -- add vector w to basis
updateFurtherElts : L V R := _
   [(lm.i*w)::V R for i in 1..maxIndex lm]
furtherElts := append (rest furtherElts, updateFurtherElts)
else
   -- the vector w lies in the span of matrix, no updating
   -- of the basis
   furtherElts := rest furtherElts
vector [row(basis, i) for i in 1..maxRowIndex basis]

standardBasisOfCyclicSubmodule (lm : L M R, v : V R) ==
   dim : NNI := #v
   standardBasis : L L R := list(entries v)
basis : M R := rowEchelon matrix list entries v
   -- normalizing the vector
   -- all these elements lie in the submodule generated by v
   furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
   -- furtherElts has elements of the generated submodule. It will
   -- will be checked whether they are in the span of the vectors
   -- computed so far. Of course we stop if we have got the whole
   -- space.
   while (^null furtherElts) and (nrows basis < #v) repeat
      w : V R := first furtherElts
      nextVector : M R := matrix list entries w -- normalizing the vector
      -- will the rank change if we add this nextVector
      -- to the basis so far computed?
      addedToBasis : M R := vertConcat(basis, nextVector)
      if rank addedToBasis ^= nrows basis then
         standardBasis := cons(entries w, standardBasis)
basis := rowEchelon addedToBasis -- add vector w to basis
   updateFurtherElts : L V R := _
      [(lm.i*w for i in 1..maxIndex lm]
furtherElts := append (rest furtherElts, updateFurtherElts)
else
   -- the vector w lies in the span of matrix, therefore
   -- no updating of matrix
   furtherElts := rest furtherElts
transpose matrix standardBasis

if R has Field then -- only because of inverse in Matrix
   -- as conditional local functions, *internal have to be here
splitInternal: (L M R, V R, B) -> L L M R
   n : I := # vector -- R-rank of representation module =
   -- degree of representation
   submodule : V V R := cyclicSubmodule (algebraGenerators, vector)
rankOfSubmodule : I := # submodule -- R-Rank of submodule
submoduleRepresentation : L M R := nil()
factormoduleRepresentation : L M R := nil()
if n ^= rankOfSubmodule then
  messagePrint " A proper cyclic submodule is found."
  if doSplitting? then -- no else !!
    submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
    factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..n]
    transitionMatrix : M R := _
      transpose completeEchelonBasis submodule
    messagePrint " Transition matrix computed"
    inverseTransitionMatrix : M R := _
      autoCoerce(inverse transitionMatrix) Union(M R,"failed")
    messagePrint " The inverse of the transition matrix computed"
    messagePrint " Now transform the matrices"
    for i in 1..maxIndex algebraGenerators repeat
      helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
      -- in order to not create extra space and regarding the fact
      -- that we only want the two blocks in the main diagonal we
      -- multiply with the aid of the local function blockMultiply
      submoduleRepresentation := cons( blockMultiply(_
        helpMatrix,transitionMatrix,submoduleIndices,n), _
        submoduleRepresentation)
      factormoduleRepresentation := cons( blockMultiply(_
        helpMatrix,transitionMatrix,factormoduleIndices,n), _
        factormoduleRepresentation)
    [reverse submoduleRepresentation, reverse _
      factormoduleRepresentation]
else -- representation is irreducible
  messagePrint " The generated cyclic submodule was not proper"
  [algebraGenerators]

irreducibilityTestInternal: (L M R, M R, B) -> L L M R
irreducibilityTestInternal(algebraGenerators,_
  singularMatrix,split?) ==
  algebraGeneratorsTranspose : L M R := [transpose _
    algebraGenerators.j for j in 1..maxIndex algebraGenerators]
  xt : M R := transpose singularMatrix
  messagePrint " We know that all the cyclic submodules generated by all"
  messagePrint " non-trivial element of the singular matrix under view are"
  messagePrint " not proper, hence Norton's irreducibility test can be done:"
  -- actually we only would need one (!) non-trivial element from
  -- the kernel of xt, such an element must exist as the transpose
  -- of a singular matrix is of course singular. Question: Can
  -- we get it more easily from the kernel of x = singularMatrix?
  kernel : L V R := nullSpace xt
  result : L L M R := _
    splitInternal(algebraGeneratorsTranspose,first kernel,split?)
if null rest result then -- this means first kernel generates
-- the whole module
if 1 = #kernel then
  messagePrint " Representation is absolutely irreducible"
else
  messagePrint " Representation is irreducible, but we don’t know "
  messagePrint " whether it is absolutely irreducible"
else
  if split? then
    messagePrint " Representation is not irreducible and it will be split:"
    -- these are the dual representations, so calculate the
    -- dual to get the desired result, i.e. "transpose inverse"
    -- improvements??
    for i in 1..maxIndex result repeat
      for j in 1..maxIndex (result.i) repeat
        mat : M R := result.i.j
        result.i.j := _
        transpose autoCoerce(inverse mat)$Union(M R,"failed")
  else
    messagePrint " Representation is not irreducible, use meatAxe to split"
    -- if "split?" then dual representation interchange factor
    -- and submodules, hence reverse
    reverse result

-- exported functions for FiniteField-s.

areEquivalent? (aG0, aG1) ==
  areEquivalent? (aG0, aG1, true, 25)

areEquivalent? (aG0, aG1, numberOfTries) ==
  areEquivalent? (aG0, aG1, true, numberOfTries)

areEquivalent? (aG0, aG1, randomelements, numberOfTries) ==
  result : B := false
  transitionM : M R := zero(1, 1)
  numberOfGenerators : NNI := #aG0
  -- need a start value for creating random matrices:
  -- if we switch to randomelements later, we take the last
  -- fingerprint.
  if randomelements then -- random should not be from I
    --randomIndex : I := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    x0 : M R := aG0.randomIndex
    x1 : M R := aG1.randomIndex
  n : NNI := #row(x0,1) -- degree of representation
foundResult := false
for i in 1..numberOfTries until foundResult repeat
  -- try to create a non-singular element of the algebra
  -- generated by "aG". If only two generators,
  -- i < 7 and not "randomelements" use Parker’s fingerprints
  -- i >= 7 create random elements recursively:
  -- \( x_{i+1} := x_i \ast mr1 + mr2 \), where mr1 and mr2 are randomly
  -- chosen elements form "aG".
  if i = 7 then randomelements := true
  if randomelements then
    --randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    x0 := x0 \ast aG0.randomIndex
    x1 := x1 \ast aG1.randomIndex
  --randomIndex := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  x0 := x0 + aG0.randomIndex
  x1 := x1 + aG1.randomIndex
  else
    x0 := fingerPrint (i, aG0.0, aG0.1 ,x0)
    x1 := fingerPrint (i, aG1.0, aG1.1 ,x1)
  -- test singularity of x0 and x1
  rk0 : NNI := rank x0
  rk1 : NNI := rank x1
  rk0 ^= rk1 =>
    messagePrint "Dimensions of kernels differ"
    foundResult := true
    result := false
  -- can assume dimensions are equal
  rk0 ^= n - 1 =>
    -- not of any use here if kernel not one-dimensional
    if randomelements then
      messagePrint "Random element in generated algebra does"
      messagePrint " not have a one-dimensional kernel"
    else
      messagePrint "Fingerprint element in generated algebra does"
      messagePrint " not have a one-dimensional kernel"
    -- can assume dimensions are equal and equal to n-1
    if randomelements then
      messagePrint "Random element in generated algebra has"
      messagePrint " one-dimensional kernel"
    else
      messagePrint "Fingerprint element in generated algebra has"
      messagePrint " one-dimensional kernel"
  kernel0 : L V R := nullSpace x0
  kernel1 : L V R := nullSpace x1
  baseChange0 : M R := standardBasisOfCyclicSubmodule(_
    aG0,kernel0.1)
  baseChange1 : M R := standardBasisOfCyclicSubmodule(_
    aG1,kernel1.1)
messagePrint "Dimensions of generated cyclic submodules differ"$
foundResult := true
result := false
-- can assume that dimensions of cyclic submodules are equal
(ncols baseChange0) = n => -- full dimension
transitionM := baseChange0 * _
    autoCoerce(inverse baseChange1)$Union(M R,"failed")
foundResult := true
result := true
for j in 1..numberOfGenerators while result repeat
    if (aG0.j*transitionM) ^= (transitionM*aG1.j) then
        result := false
        transitionM := zero(1 ,1)
        messagePrint "There is no isomorphism, as the only possible one"
        messagePrint "fails to do the necessary base change"
        -- can assume that dimensions of cyclic submodules are not "n"
        messagePrint "Generated cyclic submodules have equal, but not full"
        messagePrint "dimension, hence we can not draw any conclusion"
        -- here ends the for-loop
    if not foundResult then
        messagePrint ""
        messagePrint "Can neither prove equivalence nor inequivalence."
        messagePrint "Try again."
    else
        if result then
            messagePrint ""
            messagePrint "Representations are equivalent."
        else
            messagePrint ""
            messagePrint "Representations are not equivalent."
        transitionM

isAbsolutelyIrreducible?(aG) == isAbsolutelyIrreducible?(aG,25)

isAbsolutelyIrreducible?(aG, numberOfTries) ==
result : B := false
numberOfGenerators : NNI := #aG
-- need a start value for creating random matrices:
-- randomIndex : I := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)
x : M R := aG.randomIndex
n : NNI := #row(x,1) -- degree of representation
foundResult : B := false
for i in 1..numberOfTries until foundResult repeat
    -- try to create a non-singular element of the algebra
    -- generated by "aG", dimension of its kernel being 1.
    -- create random elements recursively:
-- $x_{i+1} := x_i \cdot r_1 + r_2$, where $r_1$ and $r_2$ are randomly
-- chosen elements from "aG".
-- randomIndex := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)
x := x \cdot aG.randomIndex
-- randomIndex := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)
x := x + aG.randomIndex
-- test whether rank of x is n-1
rk : NNI := rank x
if rk = n - 1 then
  foundResult := true
  messagePrint "Random element in generated algebra has"
messagePrint " one-dimensional kernel"
  kernel : L V R := nullSpace x
  if n=#cyclicSubmodule(aG, first kernel) then
    result := (irreducibilityTestInternal(aG,x,false)).1 ^= nil()$(L M R)
    -- result := not null? first irreducibilityTestInternal(aG,x,false) -- this doesn't
else -- we found a proper submodule
  result := false
  -- split(aG,kernel.1) -- to get the splitting
else -- not of any use here if kernel not one-dimensional
  messagePrint "Random element in generated algebra does"
messagePrint " not have a one-dimensional kernel"
-- here ends the for-loop
if not foundResult then
  messagePrint "We have not found a one-dimensional kernel so far,"
  messagePrint " as we do a random search you could try again"
--else
-- if not result then
--  messagePrint "Representation is not irreducible."
-- else
--  messagePrint "Representation is irreducible."
result

split(algebraGenerators: L M R, vector: V R) ==
  splitInternal(algebraGenerators, vector, true)

split(algebraGenerators : L M R, submodule: V V R) == -- not zero submodule
  n : NNI := #submodule.1 -- R-rank of representation module =
  -- degree of representation
  rankOfSubmodule : I := (#submodule) :: I -- R-Rank of submodule
  submoduleRepresentation : L M R := nil()
  factormoduleRepresentation : L M R := nil()
  submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
  factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..(n::I)]
  transitionMatrix : M R := _
transpose completeEchelonBasis submodule
messagePrint " Transition matrix computed"
inverseTransitionMatrix : M R :=
    autoCoerce(inverse transitionMatrix)$Union(M R,"failed")
messagePrint " The inverse of the transition matrix computed"
messagePrint " Now transform the matrices"
for i in 1..maxIndex algebraGenerators repeat
    helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
    -- in order to not create extra space and regarding the fact
    -- that we only want the two blocks in the main diagonal we
    -- multiply with the aid of the local function blockMultiply
    submoduleRepresentation := cons( blockMultiply( _
        helpMatrix,transitionMatrix,submoduleIndices,n), _
        submoduleRepresentation)
    factormoduleRepresentation := cons( blockMultiply( _
        helpMatrix,transitionMatrix,factormoduleIndices,n), _
        factormoduleRepresentation)
    cons(reverse submoduleRepresentation, list( reverse _
        factormoduleRepresentation)::(L L M R))

-- the following is "under" "if R has Field", as there are compiler
-- problems with conditionally defined local functions, i.e. it
-- doesn't know, that "FiniteField" has "Field".

-- we are scanning through the vectorspaces
if (R has Finite) and (R has Field) then
    meatAxe(algebraGenerators, randomelements, numberOfTries, _
        maxTests) ==
    numberOfGenerators : NNI := #algebraGenerators
    result : L L M R := nil()$(L L M R)
    q : PI := size()$R:PI
    -- need a start value for creating random matrices:
    -- if we switch to randomelements later, we take the last
    -- fingerprint.
    if randomelements then -- random should not be from I
        --randomIndex : I := randnum numberOfGenerators
        randomIndex := 1+(random()$Integer rem numberOfGenerators)
        x : M R := algebraGenerators.randomIndex
    foundResult : B := false
    for i in 1..numberOfTries until foundResult repeat
        -- try to create a non-singular element of the algebra
        -- generated by "algebraGenerators". If only two generators,
        -- i < 7 and not "randomelements" use Parker's fingerprints
        -- i >= 7 create random elements recursively:
        -- x_{i+1} := x_{i} * mr1 + mr2, where mr1 and mr2 are randomly
        -- chosen elements form "algebraGenerators".
        if i = 7 then randomelements := true
if randomelements then
  --randomIndex := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  x := x * algebraGenerators.randomIndex
  --randomIndex := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  x := x + algebraGenerators.randomIndex
else
  x := fingerPrint (i, algebraGenerators.1, algebraGenerators.2, x)
  -- test singularity of x
  n : NNI := #row(x, 1) -- degree of representation
  if (rank x) ^= n then -- x singular
    if randomelements then
      messagePrint "Random element in generated algebra is singular"
    else
      messagePrint "Fingerprint element in generated algebra is singular"
    kernel : L V R := nullSpace x
    -- the first number is the maximal number of one dimensional
    -- subspaces of the kernel, the second is a user given
    -- constant
    numberOfOneDimSubspacesInKernel : I := (q**(#kernel)-1)quo(q-1)
    numberOfTests : I := _
    min(numberOfOneDimSubspacesInKernel, maxTests)
    for j in 1..numberOfTests repeat
      --we create an element in the kernel, there is a good
      --probability for it to generate a proper submodule, the
      --called "split" does the further work:
      result := _
      split(algebraGenerators,scanOneDimSubspaces(kernel,j))
      -- we had "not null rest result" directly in the following
      -- if .. then, but the statement there foundResult := true
      -- didn't work properly
      foundResult := not null rest result
      if foundResult then
        leave -- inner for-loop
        -- finish here with result
      else -- no proper submodule
        -- we were not successful, i.e gen. submodule was
        -- not proper, if the whole kernel is already scanned,
        -- Norton's irreducibility test is used now.
        if (j+1)>numberOfOneDimSubspacesInKernel then
          -- we know that all the cyclic submodules generated
          -- by all non-trivial elements of the kernel are proper.
          foundResult := true
          result : L L M R := irreducibilityTestInternal (_
            algebraGenerators,x,true)
          leave -- inner for-loop
        -- here ends the inner for-loop
      else -- x non-singular
if randomelements then
  messagePrint "Random element in generated algebra is non-singular"
else
  messagePrint "Fingerprint element in generated algebra is non-singular"

-- here ends the outer for-loop
if not foundResult then
  result : L L M R := [nil()$(L M R), nil()$(L M R)]
  messagePrint ""
  messagePrint "Sorry, no result, try meatAxe(...,true)"
  messagePrint " or consider using an extension field."
  result

meatAxe (algebraGenerators) ==
  meatAxe(algebraGenerators, false, 25, 7)

meatAxe (algebraGenerators, randomElements?) ==
  randomElements? => meatAxe (algebraGenerators, true, 25, 7)
  meatAxe(algebraGenerators, false, 6, 7)

meatAxe (algebraGenerators:L M R, numberOfTries:PI) ==
  meatAxe (algebraGenerators, true, numberOfTries, 7)

scanOneDimSubspaces(basis,n) ==
  "dimension" of subspace generated by "basis"
  dim : NNI := #basis
  "dimension of the whole space:
  nn : NNI := #(basis.1)
  q : NNI := size()$R
  -- number of all one-dimensional subspaces:
  nred : I := n rem ((q**dim -1) quo (q-1))
  pos : I := nred
  i : I := 0
  for i in 0..dim-1 while nred >= 0 repeat
    pos := nred
    nred := nred - (q**i)
    i := if i = 0 then 0 else i-1
    coefficients : V R := new(dim,0$R)
    coefficients.(dim-i) := 1$R
    iR : L I := wholeRagits(pos::RADIX q)
    for j in 1..(maxIndex iR) repeat
      coefficients.(dim-((#iR)::I) +j) := index((iR.j+(q::I))::PI)$R
    result : V R := new(nn,0)
    for i in 1..maxIndex coefficients repeat
      newAdd : V R := coefficients.i * basis.i
      for j in 1..nn repeat
result.j := result.j + newAdd.j
result

---

— REP2.dotabb —

"REP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP2"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"REP2" -> "IVECTOR"

---

package RESLATC ResolveLatticeCompletion

--- ResolveLatticeCompletion.input ---

)set break resume
)sys rm -f ResolveLatticeCompletion.output
)spool ResolveLatticeCompletion.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ResolveLatticeCompletion
--E 1

)spool
)lisp (bye)

---

— ResolveLatticeCompletion.help —

====================================================================
ResolveLatticeCompletion examples
====================================================================

This package provides coercions for the special types Exit and Void.

See Also:
 o )show ResolveLatticeCompletion
ResolveLatticeCompletion (RESLATC)

Exports:

coerce

— package RESLATC ResolveLatticeCompletion —

)abbrev package RESLATC ResolveLatticeCompletion
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides coercions for the special types \spadtype{Exit}
++ and \spadtype{Void}.

ResolveLatticeCompletion(S: Type): with
  coercex: S -> Void
  ++ coerce(s) throws all information about s away.
  ++ This coercion allows values of any type to appear
  ++ in contexts where they will not be used.
  ++ For example, it allows the resolution of different types in
  ++ the \spad{then} and \spad{else} branches when an \spad{if}
  ++ is in a context where the resulting value is not used.
  coercex: Exit -> S
  ++ coerce(e) is never really evaluated. This coercion is
  ++ used for formal type correctness when a function will not
  ++ return directly to its caller.

== add
  coercex(s: S): Void == void()
  coercex(e: Exit): S ==
      error "Bug: Should not be able to obtain value of type Exit"
package RETSOL RetractSolvePackage

 RetractSolvePackage.examples

RetractSolvePackage is an interface to SystemSolvePackage that attempts to retract the coefficients of the equations before solving.

See Also:
- )show RetractSolvePackage
RetractSolvePackage (RETSOL)

Exports:
solveRetract

--- package RETSOL RetractSolvePackage ---

)abbrev package RETSOL RetractSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 31 October 1991
++ Description:
++ RetractSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving.

RetractSolvePackage(Q, R): Exports == Implementation where
  Q: IntegralDomain
  R: Join(IntegralDomain, RetractableTo Q)

  PQ ==> Polynomial Q
  FQ ==> Fraction PQ
  SY ==> Symbol
  P ==> Polynomial R
  F ==> Fraction P
  EQ ==> Equation
  SSP ==> SystemSolvePackage

Exports == with
  solveRetract: (List P, List SY) -> List List EQ F
++ solveRetract(lp,lv) finds the solutions of the list lp of
++ rational functions with respect to the list of symbols lv.
++ The function tries to retract all the coefficients of the equations
++ to Q before solving if possible.

Implementation == add
  LEQ2F : List EQ FQ -> List EQ F
\[ \text{FQ2F} : \text{FQ} \rightarrow \text{F} \]
\[ \text{PQ2P} : \text{PQ} \rightarrow \text{P} \]
\[ \text{QIfCan} : \text{List P} \rightarrow \text{Union(List FQ, "failed")} \]
\[ \text{PQIfCan} : \text{P} \rightarrow \text{Union(FQ, "failed")} \]

\[ \text{PQ2P} \ p \ = \ \text{map}((q1:Q):R \mapsto q1::R)\text{PolynomialFunctions2}(Q, R) \]
\[ \text{FQ2F} \ f \ = \ \text{PQ2P numer f / PQ2P denom f} \]
\[ \text{LEQQ2F} \ l \ = \ [\text{equation(FQ2F lhs eq, FQ2F rhs eq) for eq in l}] \]

\[ \text{solvRetract(1p, 1v) ==} \]
\[ \ (u := \text{QIfCan lp}) \text{ case "failed" =>} \]
\[ \quad \text{solve([p::F for p in 1p]\text{List(F)}, 1v)\text{SSP(R)}} \]
\[ \quad [\text{LEQQ2F l for 1 in solve(u::List(FQ), 1v)\text{SSP(Q)}]} \]

\[ \text{QIfCan l ==} \]
\[ \quad \text{ans:List(FQ) := empty() for p in l repeat} \]
\[ \quad \quad (u := \text{PQIfCan p}) \text{ case "failed" => return "failed"} \]
\[ \quad \quad \text{ans := concat(u::FQ, ans)} \]
\[ \quad \text{ans} \]

\[ \text{PQIfCan p ==} \]
\[ \quad (u := \text{mainVariable p}) \text{ case "failed" =>} \]
\[ \quad \quad (r := \text{retractIfCan(ground p)\text{Union(Q,"failed")}}) \text{ case Q => r::Q::PQ::FQ "failed"} \]
\[ \quad \quad \text{up := univariate(p, s := u::SY)} \]
\[ \quad \quad \text{ans:FQ := 0 while up ^= 0 repeat} \]
\[ \quad \quad \quad (v := \text{PQIfCan leadingCoefficient up}) \text{ case "failed" => return "failed"} \]
\[ \quad \quad \quad \text{ans := ans + monomial(1, s, degree up)\text{PQ * (v::FQ)}} \]
\[ \quad \quad \quad \text{up := reductum up} \]
\[ \quad \text{ans} \]

— RETSOL.dotabb —

"RETSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RETSOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RETSOL" -> "PFECAT"
package RFP RootsFindingPackage

--- RootsFindingPackage.input ---

)set break resume
)sys rm -f RootsFindingPackage.output
)spool RootsFindingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RootsFindingPackage
--R
--R RootsFindingPackage(K: Field) is a package constructor
--R Abbreviation for RootsFindingPackage is RFP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFP
--R
--R------------------------------- Operations --------------------------------
--R foundZeroes : () -> List(K)
--R distinguishedCommonRootsOf : (List(SparseUnivariatePolynomial(K)),K) -> Record(zeros: List(K),extDegree: Integer)
--R distinguishedRootsOf : (SparseUnivariatePolynomial(K),K) -> Record(zeros: List(K),extDegree: Integer)
--R setFoundZeroes : List(K) -> List(K)
--R
--E 1

)spool
)lisp (bye)

---

--- RootsFindingPackage.help ---

====================================================================
RootsFindingPackage examples
====================================================================

This pacakage finds all the roots of a polynomial. If the constant field is not large enough then it returns the list of found zeros and the degree of the extension need to find the other roots missing. If the return degree is 1 then all the roots have been found. If 0 is return for the extension degree then there are an infinite number of zeros, that is you ask for the zeroes of 0. In the case of infinite field a list of all found zeros is kept and for each other call of a function that finds zeroes, a check is made on that list; this is to keep a kind of "canonical" representation of the elements.
See Also:
- )show RootsFindingPackage

---

RootsFindingPackage (RFP)

RFP

PACRATC

Exports:

distinguishedCommonRootsOf distinguishedRootsOf foundZeroes setFoundZeroes

— package RFP RootsFindingPackage —

)abbrev package RFP RootsFindingPackage
++ Authors: G. Hache
++ Date Created: 6 Oct 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This pacackage finds all the roots of a polynomial. If the constant field is
++ not large enough then it returns the list of found zeros and the degree
++ of the extension need to find the other roots missing. If the return
++ degree is 1 then all the roots have been found. If 0 is return
++ for the extension degree then there are an infinite number of zeros,
++ that is you ask for the zeroes of 0. In the case of infinite field
++ a list of all found zeros is kept and for each other call of a function
++ that finds zeroes, a check is made on that list; this is to keep
++ a kind of “canonical” representation of the elements.

RootsFindingPackage(K):P==T where
  K:Field
  LIST ==> List
  INT ==> Integer
  NNI ==> NonNegativeInteger
  MFINFACT ==> MultFiniteFactorize
PACKAGE RFP ROOTSFINDINGPACKAGE

FFFACTSE ==> FiniteFieldFactorizationWithSizeParseBySideEffect
SUP    ==> SparseUnivariatePolynomial
REC    ==> Record(zeros:List(K),extDegree:INT)

P== with

  distinguishedRootsOf: (SUP(K),K) -> REC
  ++ distinguishedRootsOf returns a record consisting of a list of zeros
  ++ of the input polynomial followed by the smallest extension degree
  ++ needed to find all the zeros. If K has
  ++ \spad{PseudoAlgebraicClosureOfFiniteFieldCategory} or
  ++ \spad{PseudoAlgebraicClosureOfRationalNumberCategory} then
  ++ a root is created for each irreducible factor, and only these
  ++ roots are returns and not their conjugate.

distinguishedCommonRootsOf: (List SUP(K),K) -> REC
  ++ distinguishedCommonRootsOf returns the common zeros of a list of
  ++ polynomial. It returns a record as in distinguishedRootsOf. If 0
  ++ is returned as extension degree then there are an infinite number
  ++ of common zeros (in this case, the polynomial 0 was given in the
  ++ list of input polynomials).

foundZeroes: () -> List K
  ++ foundZeroes returns the list of already
  ++ found zeros by the functions
  ++ distinguishedRootsOf and
  ++ distinguishedCommonRootsOf.

setFoundZeroes: List K -> List K
  ++ setFoundZeroes sets the list of foundZeroes to the given one.

T== add
  -- signature of local function
zeroOfLinearPoly: SUP(K) -> K
  -- local variable
listOfAllZeros:List(K):=empty()

foundZeroes==listOfAllZeros

if K has PseudoAlgebraicClosureOfPerfectFieldCategory then
  distinguishedRootsOf(polyZero, theExtension) ==
  --PRECONDITION: setExtension! is called in K to set the extension to
  --the extension of factorization
  zero?(polyZero) =>
  [empty(),0]
  listOfZeros:List(K):=distinguishedRootsOf(polyZero, theExtension)$K
  [listOfZeros,1]

if K has FiniteFieldCategory and
  "(K has PseudoAlgebraicClosureOfFiniteFieldCategory) then
distinguishedRootsOf(polyZero,dummy) ==
  zero?(polyZero) => [empty(),0]
factorpolyZero:=factor(polyZero)$FFFACTSE(K,SUP(K))
listOfFactor:= [pol.fctr for pol in listOfFactor]
listFact:= [pol.fctr for pol in listOfFactor]
degExt:INT:=
  lcm([degree(poly) for poly in listFact]) pretend LIST(INT))
listOfZeros:List(K):=removeDuplicates_
  [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
  [listOfZeros,degExt]

if K has QuotientFieldCategory( Integer ) and _
  ^K has PseudoAlgebraicClosureOfRationalNumberCategory) then
distinguishedRootsOf(polyZero,dummy) ==
  zero?(polyZero) => [empty(),0]
factorpolyZero:=factor(polyZero)$RationalFactorize( SUP(K) )
listOfFactor:= [pol.fctr for pol in listOfFactor]
listFact:= [pol.fctr for pol in listFact]
degExt:INT:=
  lcm([degree(poly) for poly in listFact]) pretend LIST(INT))
listOfZeros:List(K):=removeDuplicates_
  [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
  [listOfZeros,degExt]

distinguishedCommonRootsOf(listOfPoly1,theExtension)==
  listOfPoly:List(SUP(K)):= [pol for pol in listOfPoly1 | ^zero?(pol)]
  empty?(listOfPoly) ==> [empty(),0]
  reco:= distinguishedRootsOf(gcd(listOfPoly),theExtension)
  listOfZeros:= reco.zeros
  degExt:INT:= reco.extDegree
  [listOfZeros,degExt]

zeroOfLinearPoly(pol)==
  "one?(degree(pol)) => error "the polynomial is not linear"
  listCoeff:List(K):=coefficients(pol)
  one?(#listCoeff) => 0
  - last(listCoeff) / first(listCoeff)

setFoundZeros(setlist)==
  oldList0fAllZeros:= copy list0fAllZeros
  list0fAllZeros:=setlist
  oldList0fAllZeros

—— RFP.dotabb ——
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"PACRATC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PACRATC"]
"RFP" -> "PACRATC"
Chapter 20

Chapter S

package SAERFFC SAERationalFunctionAlgFactor

— SAERationalFunctionAlgFactor.input —

)set break resume
)sys rm -f SAERationalFunctionAlgFactor.output
)spool SAERationalFunctionAlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SAERationalFunctionAlgFactor
--E 1

)spool
)lisp (bye)

— SAERationalFunctionAlgFactor.help —

====================================================================
SAERationalFunctionAlgFactor examples
====================================================================

Factorization of univariate polynomials with coefficients in an
algebraic extension of Fraction Polynomial Integer.

See Also:

3365
o )show SAERationalFunctionAlgFactor

---

SAERationalFunctionAlgFactor (SAERFFC)

Exports:
factor

--- package SAERFFC SAERationalFunctionAlgFactor ---

)abbrev package SAERFFC SAERationalFunctionAlgFactor
++ Author: Patrizia Gianni
++ Description:
++ Factorization of univariate polynomials with coefficients in an
++ algebraic extension of \spadtype{Fraction Polynomial Integer}.

SAERationalFunctionAlgFactor(UP, SAE, UPA): Exports == Implementation where

UP : UnivariatePolynomialCategory Fraction Polynomial Integer
SAE : Join(Field, CharacteristicZero,
            MonogenicAlgebra(Fraction Polynomial Integer, UP))
UPA: UnivariatePolynomialCategory SAE

Exports ==> with
  factor: UPA -> Factored UPA
  ++ factor(p) returns a prime factorisation of p.

Implementation ==> add
  factor q ==
    factor(q, factor$RationalFunctionFactor(UP))
  )$InnerAlgFactor(Fraction Polynomial Integer, UP, SAE, UPA)

---
package FORMULA1 ScriptFormulaFormat1

ScriptFormulaFormat1 examples

ScriptFormulaFormat1 provides a utility coercion for changing to SCRIPT formula format anything that has a coercion to the standard output format.

See Also:
- )show ScriptFormulaFormat1
Exports:
coerce
package SEGBIND2 SegmentBindingFunctions2

— SegmentBindingFunctions2.input —

)set break resume
)sys rm -f SegmentBindingFunctions2.output
)spool SegmentBindingFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SegmentBindingFunctions2
--E 1

)spool
)lisp (bye)

— SegmentBindingFunctions2.help —

====================================================================
SegmentBindingFunctions2 examples
====================================================================

This package provides operations for mapping functions onto SegmentBinding.

See Also:
  o )show SegmentBindingFunctions2

—
SegmentBindingFunctions2 (SEGBIND2)

Exports:

map

— package SEGBIND2 SegmentBindingFunctions2 —

)abbrev package SEGBIND2 SegmentBindingFunctions2
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations for mapping functions onto
++ \spadtype{SegmentBinding}s.

SegmentBindingFunctions2(R:Type, S:Type): with
  map: (R -> S, SegmentBinding R) -> SegmentBinding S
    ++ map(f,v=a..b) returns the value given by \spad{v=f(a)..f(b)}.
    == add
    map(f, b) ==
      equation(variable b, map(f, segment b)$SegmentFunctions2(R, S))

— SEGBIND2.dotabb —

"SEGBIND2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SEGBIND2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"SEGBIND2" -> "TYPE"
package SEG2 SegmentFunctions2

--- SegmentFunctions2.input ---

)set break resume
)sys rm -f SegmentFunctions2.output
)spool SegmentFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SegmentFunctions2
--E 1

)spool
)lisp (bye)

---

--- SegmentFunctions2.help ---

====================================================================
SegmentFunctions2 examples
====================================================================

This package provides operations for mapping functions onto segments.

See Also:
  o )show SegmentFunctions2

---
SegmentFunctions2 (SEG2)

Exports:
map

— package SEG2 SegmentFunctions2 —

)abbrev package SEG2 SegmentFunctions2
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations for mapping functions onto segments.

SegmentFunctions2(R:Type, S:Type): public == private where
public ==> with
map: (R -> S, Segment R) -> Segment S
++ map(f,l..h) returns a new segment \spad{f(l)..f(h)}.

if R has OrderedRing then
map: (R -> S, Segment R) -> List S
++ map(f,s) expands the segment s, applying \spad{f} to each
++ value. For example, if \spad{s = l..h by k}, then the list
++ \spad{[f(l), f(l+k),..., f(lN)]} is computed, where
++ \spad{lN <= h < lN+k}.

private ==> add
map(f : R->S, r : Segment R): Segment S ==
SEGMENT(f lo r,f hi r)$Segment(S)

if R has OrderedRing then
map(f : R->S, r : Segment R): List S ==
lr := nil()$List(S)
l := lo r
h := hi r
inc := (incr r)::R
if inc > 0 then
while l <= h repeat
lr := concat(f(l), lr)
1 := 1 + inc
else
    while l >= h repeat
        lr := concat(f(l), lr)
        l := l + inc
    reverse_! lr

— SEG2.dotabb —

"SEG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SEG2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"SEG2" -> "FLAGG"

package SAEFACT SimpleAlgebraicExtensionAlgFactor

— SimpleAlgebraicExtensionAlgFactor.input —

)set break resume
)sys rm -f SimpleAlgebraicExtensionAlgFactor.output
)spool SimpleAlgebraicExtensionAlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SimpleAlgebraicExtensionAlgFactor
--E 1

)spool
)lisp (bye)

— SimpleAlgebraicExtensionAlgFactor.help —

====================================================================
SimpleAlgebraicExtensionAlgFactor examples
====================================================================
Factorization of univariate polynomials with coefficients in an algebraic extension of the rational numbers (\textit{Fraction Integer}).

See Also:
- \texttt{\textbackslash )show SimpleAlgebraicExtensionAlgFactor}

---

\textbf{SimpleAlgebraicExtensionAlgFactor (SAEFACT)}

- \texttt{RFFACT}
- \texttt{ALIST}

Exports:

\begin{verbatim}
factor
\end{verbatim}

--- package SAEFACT SimpleAlgebraicExtensionAlgFactor ---

\texttt{\textbackslash abbrev package SAEFACT SimpleAlgebraicExtensionAlgFactor}

++ Author: Patrizia Gianni
++ Description:
++ Factorization of univariate polynomials with coefficients in an algebraic extension of the rational numbers (\textit{\textbackslash spadtype{Fraction Integer}}).

\texttt{SimpleAlgebraicExtensionAlgFactor(UP,SAE,UPA):Exports==Implementation where}
\begin{verbatim}
UP : UnivariatePolynomialCategory Fraction Integer
SAE : Join(Field, CharacteristicZero, MonogenicAlgebra(Fraction Integer, UP))
UPA: UnivariatePolynomialCategory SAE
\end{verbatim}

Exports ==>
\begin{verbatim}
factor: UPA -> Factored UPA
\end{verbatim}
++ factor(p) returns a prime factorisation of p.

Implementation ==> add
factor q ==
factor(q, factor$RationalFactorize(UP)
 )$InnerAlgFactor(Fraction Integer, UP, SAE, UPA)

— SAEFACT.dotabb —

"SAEFAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SAEFAC"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"SAEFAC" -> "MONOGEN"

package SIMPAN SimplifyAlgebraicNumberConvertPackage

— SimplifyAlgebraicNumberConvertPackage.input —

)set break resume
)sys rm -f SimplifyAlgebraicNumberConvertPackage.output
)spool SimplifyAlgebraicNumberConvertPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show SimplifyAlgebraicNumberConvertPackage
-- E 1

)spool
)lisp (bye)

— SimplifyAlgebraicNumberConvertPackage.help —

====================================================================
SimplifyAlgebraicNumberConvertPackage examples
====================================================================

Package to allow simplify to be called on AlgebraicNumbers
by converting to EXPR(INT)
See Also:
  o )show SimplifyAlgebraicNumberConvertPackage

---

SimplifyAlgebraicNumberConvertPackage (SIMPAN)

Exports:
simplify

--- package SIMPAN SimplifyAlgebraicNumberConvertPackage ---

)abbrev package SIMPAN SimplifyAlgebraicNumberConvertPackage
++ Description:
++ Package to allow simplify to be called on AlgebraicNumbers
++ by converting to EXPR(INT)

SimplifyAlgebraicNumberConvertPackage(): with
  simplify: AlgebraicNumber -> Expression(Integer)
  ++ simplify(an) applies simplifications to an
  == add
  simplify(a:AlgebraicNumber) ==
  simplify(a::Expression(Integer))$TranscendentalManipulations(Integer, Expression Integer)

--- SIMPAN.dotabb ---

"SIMPAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIMPAN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SIMPAN" -> "FS"

package SMITH SmithNormalForm

— SmithNormalForm.input —

)set break resume
)sys rm -f SmithNormalForm.output
)spool SmithNormalForm.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SmithNormalForm
--E 1

)spool
)lisp (bye)

— SmithNormalForm.help —

====================================================================
SmithNormalForm examples
====================================================================

SmithNormalForm is a package which provides some standard canonical forms for matrices.

See Also:
o )show SmithNormalForm
SmithNormalForm (SMITH)

Exports:
completeHermite completeSmith diophantineSystem hermite smith

--- package SMITH SmithNormalForm ---

)abbrev package SMITH SmithNormalForm
++ Author: Patrizia Gianni
++ Date Created: October 1992
++ Description:
++ \spadtype{SmithNormalForm} is a package
++ which provides some standard canonical forms for matrices.

SmithNormalForm(R,Row,Col,M) : Exports == Implementation where

R : EuclideanDomain
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)

I ==> Integer
NNI ==> NonNegativeInteger
HermiteForm ==> Record(Hermite:M,eqMat:M)
SmithForm ==> Record(Smith : M, leftEqMat : M, rightEqMat : M)
PartialV ==> Union(Col, "failed")
Both ==> Record(particular: PartialV, basis: List Col)

Exports == with
hermite : M -> M
  ++ \spad{hermite(m)} returns the Hermite normal form of the
  ++ matrix m.
completeHermite : M -> HermiteForm
  ++ \spad{completeHermite} returns a record that contains
  ++ the Hermite normal form H of the matrix and the equivalence matrix
  ++ U such that U*\text{m} = H
smith : M -> M
++ \spad{smith(m)} returns the Smith Normal form of the matrix m.

++ \spad{completeSmith} returns a record that contains
++ the Smith normal form H of the matrix and the left and right
++ equivalence matrices U and V such that U*m*v = H

++ \spad{diophantineSystem(A,B)} returns a particular integer solution and
++ an integer basis of the equation \spad{AX = B}.

Implementation == add

MATCAT1 ==> MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Col2,M2)
MATCAT2 ==> MatrixCategoryFunctions2(QF,Row2,Col2,M2,R,Row,Col,M)
QF  ==> Fraction R
Row2 ==> Vector QF
Col2 ==> Vector QF
M2   ==> Matrix QF

------ Local Functions ------
ellRow1 : (M,I,I) -> M
ellRow2 : (M,R,I,I) -> M
ellColumn2 : (M,R,I,I) -> M
isDiagonal? : M -> Boolean
ijDivide : (SmithForm ,I,I) -> SmithForm
lastStep : SmithForm -> SmithForm

test1 : (M,Col,NNI) -> Union(NNI , "failed")
test2 : (M, Col,NNI,NNI) -> Union( Col, "failed")

-- inconsistent system : case 0 = c --
test1(sm:M,b:Col,m1 : NNI) : Union(NNI , "failed") ==
  km:=m1
  while zero? sm(km,km) repeat
    if not zero?(b(km)) then return "failed"
    km:= (km - 1) :: NNI
  km

if Col has shallowlyMutable then

  test2(sm : M ,b : Col , n1:NNI,dk:NNI) : Union( Col, "failed") ==
  -- test divisibility --
  sol:Col := new(n1,0)
  for k in 1..dk repeat
    if (c:=(b(k) exquo sm(k,k))) case "failed" then return "failed"
    sol(k):= c::R
  sol

-- test if the matrix is diagonal or pseudo-diagonal --

isDiagonal?(m : M) : Boolean ==
  m1:= nrows m
  n1:= ncols m
  for i in 1..m1 repeat
for \( j \) in 1..n1 \( | \) (\( j \neq i \)) repeat
  if not zero?\((m(i,j))\) then return false
true

-- elementary operation of first kind: exchange two rows --
elRow1(m:M,i:I,j:I) : M ==
vec:=row(m,i)
setRow!(m,i,row(m,j))
setRow!(m,j,vec)
m

-- elementary operation of second kind: add to row \( i \) --
  -- \( a * \text{row} j (i \neq j) \) --
elRow2(m : M,a:R,i:I,j:I) : M ==
vec:=map(x +-> a*x,row(m,j))
vec:=map("++",row(m,i),vec)
setRow!(m,i,vec)
m

-- elementary operation of second kind: add to column \( i \) --
  -- \( a * \text{column} j (i \neq j) \) --
elColumn2(m : M,a:R,i:I,j:I) : M ==
vec:=map(x +-> a*x,column(m,j))
vec:=map("++",column(m,i),vec)
setColumn!(m,i,vec)
m

-- modify SmithForm in such a way that the term \( m(i,i) \) --
  -- divides the term \( m(j,j) \). \( m \) is diagonal --
ijDivide(sf : SmithForm , i : I,j : I) : SmithForm ==
m:=sf.Smith
mii:=m(i,i)
mjj:=m(j,j)
extGcd:=extendedEuclidean(mii,mjj)
d := extGcd.generator
mii:=(mii exquo d)::R
mjj := (mjj exquo d) :: R
  -- add to row \( j \) extGcd.coef1*row \( i \) --
1Mat:=elRow2(sf.leftEqMat,extGcd.coef1,j,i)
  -- switch rows \( i \) and \( j \) --
1Mat:=elRow1(1Mat,i,j)
  -- add to row \( j \) -mii*row \( i \) --
1Mat := elRow2(1Mat,-mii,j,i)
  -- multiply by -1 column \( j \) --
1Mat := ijModify(mii,mjj,extGcd.coef1,extGcd.coef2,sf.leftEqMat,i,j)
m(j,j):= m(i,i) * mjj
m(i,i):= d
  -- add to column \( i \) extGcd.coef2 * column \( j \) --
rMat := elColumn2(sf.rightEqMat,extGcd.coef2,i,j)
  -- add to column \( j \) -mjj*column \( i \) --
rMat:=elColumn2(rMat,-mjj,j,i)
  -- multiply by -1 column \( \) --
setColumn!(rMat,j,map(x +-> -1*x,column(rMat,j)))
[m,1Mat,rMat]

-- given a diagonal matrix compute its Smith form --
lastStep(sf : SmithForm) : SmithForm ==
m := sf.Smith
m1 := min(nrows m, ncols m)
for i in 1..m1 while (mii := m(i,i)) ^= 0 repeat
  for j in i+1..m1 repeat
    if (m(j,j) exquo mii) case "failed" then return
      lastStep(ijDivide(sf,i,j))
  end repeat
sf

-- given m and t row-equivalent matrices, with t in upper triangular --
-- form compute the matrix u such that u*m=t --
findEqMat(m : M,t : M) : Record(Hermite : M, eqMat : M) ==
m1 := nrows m
n1 := ncols m
"and"/[zero? t(m1,j) for j in 1..n1] => -- there are 0 rows
  if "and"/[zero? t(1,j) for j in 1..n1] then return [m,scalarMatrix(m1,1)] -- m is the zero matrix
mm := horizConcat(m,scalarMatrix(m1,1))
mmh := rowEchelon mm
[subMatrix(mmh,1,m1,1,n1), subMatrix(mmh,1,m1,n1+1,n1+m1)]
u : M := zero(m1,m1)
j := 1
while t(1,j)=0 repeat j:=j+1 -- there are 0 columns
t1 := copy t
mm := copy m
if j>1 then
  t1 := subMatrix(t,1,m1,j,n1)
  mm := subMatrix(m,1,m1,j,n1)
t11 := t1(1,1)
for i in 1..m1 repeat
  u(i,1) := (mm(i,1) exquo t11) :: R
for j in 2..m1 repeat
  j0 := j
  while zero?(tjj := t(1,j0)) repeat j0:=j0+1
  u(i,j) :=
    ((mm(i,j0)-("+"/[u(i,k)*t1(k,j0) for k in 1..(j-1)])) exquo tjj) :: R
ui : M2 := map(x +-> x::QF,u)$$MATCAT1
[t,map(retract$QF,(inverse ui)::M2)$MATCAT2]

--- Hermite normal form of m ---
hermite(m:M) : M == rowEchelon m

-- Hermite normal form and equivalence matrix ---
completeHermite(m : M) : Record(Hermite : M, eqMat : M) ==
findEqMat(m,rowEchelon m)
\begin{verbatim}

smith(m : M) : M == completeSmith(m).Smith

completeSmith(m : M) : Record(Smith : M, leftEqMat : M, rightEqMat : M) ==
  cm1:=completeHermite m
  leftm:=cm1.eqMat
  m1:=cm1.Hermite
  isDiagonal? m1 => lastStep([m1,leftm,scalarMatrix(ncols m,1)])
  nr:=nrows m
  cm1:=completeHermite transpose m1
  rightm:= transpose cm1.eqMat
  m1:=cm1.Hermite
  isDiagonal? m1 =>
    cm2:=lastStep([m1,leftm,rightm])
    nrows(m:=cm2.Smith) = nr => cm2
    [transpose m,cm2.leftEqMat, cm2.rightEqMat]
  cm2:=completeSmith m1
  cm2:=lastStep([cm2.Smith,transpose(cm2.rightEqMat)*leftm, rightm*transpose(cm2.leftEqMat)])
  nrows(m:=cm2.Smith) = nr => cm2
  [transpose m, cm2.leftEqMat, cm2.rightEqMat]

-- Find the solution in R of the linear system mX = b --
diophantineSystem(m : M, b : Col) : Both ==
  sf:=completeSmith m
  sm:=sf.Smith
  m1:=nrows sm
  lm:=sf.leftEqMat
  b1:Col:= lm* b
  (t1:=test1(sm,b1,m1)) case "failed" => ["failed",empty()]
  dk:=t1 :: NNI
  n1:=ncols sm
  (t2:=test2(sm,b1,n1,dk)) case "failed" => ["failed",empty()]
  rm := sf.rightEqMat
  sol:=rm*(t2 :: Col) -- particular solution
  dk = n1 => [sol,list new(n1,0)]
  lsol:List Col := [column(rm,i) for i in (dk+1)..n1]
  [sol,lsol]

-- SMITH.dotabb --

"SMITH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SMITH"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SMITH" -> "PFECAT"

\end{verbatim}
package SCACHE SortedCache

--- SortedCache.input ---

)set break resume
)sys rm -f SortedCache.output
)spool SortedCache.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SortedCache
--E 1

)spool
)lisp (bye)

--- SortedCache.help ---

====================================================================
SortedCache examples
====================================================================

A sorted cache of a cachable set $S$ is a dynamic structure that keeps the elements of $S$ sorted and assigns an integer to each element of $S$ once it is in the cache. This way, equality and ordering on $S$ are tested directly on the integers associated with the elements of $S$, once they have been entered in the cache.

See Also:
o )show SortedCache
SortedCache (SCACHE)

Exports:
cache  clearCache  enterInCache

— package SCACHE SortedCache —

)abbrev package SCACHE SortedCache
++ Author: Manuel Bronstein
++ Date Created: 31 Oct 1988
++ Date Last Updated: 14 May 1991
++ Description:
++ A sorted cache of a cachable set S is a dynamic structure that
++ keeps the elements of S sorted and assigns an integer to each
++ element of S once it is in the cache. This way, equality and ordering
++ on S are tested directly on the integers associated with the elements
++ of S, once they have been entered in the cache.

SortedCache(S:CachableSet): Exports == Implementation where
  N ==> NonNegativeInteger
  DIFF ==> 1024

Exports ==> with
  clearCache : () -> Void
    ++ clearCache() empties the cache.
  cache : () -> List S
    ++ cache() returns the current cache as a list.
  enterInCache: (S, S -> Boolean) -> S
    ++ enterInCache(x, f) enters x in the cache, calling \spad{f(y)} to
    ++ determine whether x is equal to y. It returns x with an integer
    ++ associated with it.
  enterInCache: (S, (S, S) -> Integer) -> S
    ++ enterInCache(x, f) enters x in the cache, calling \spad{f(x, y)} to
    ++ determine whether \spad{x < y (f(x,y) < 0), x = y (f(x,y) = 0)}, or
    ++ \spad{x > y (f(x,y) > 0)}.
    ++ It returns x with an integer associated with it.
Implementation ==> add

shiftCache : (List S, N) -> Void
insertInCache: (List S, List S, S, N) -> S

cach := [nil()]$Record(cche:List S)

cache() == cach.cche

shiftCache(1, n) ==
    for x in l repeat setPosition(x, n + position x)
    void

clearCache() ==
    for x in cache repeat setPosition(x, 0)
    cach.cche := nil()
    void

enterInCache(x:S, equal?:S -> Boolean) ==
    scan := cache()
    while not null scan repeat
        equal?(y := first scan) =>
            setPosition(x, position y)
            return y
        scan := rest scan
    setPosition(x, 1 + #cache())
    cach.cche := concat(cache(), x)
    x

enterInCache(x:S, triage:(S, S) -> Integer) ==
    scan := cache()
    pos:N:= 0
    for i in 1..#scan repeat
        zero?(n := triage(x, y := first scan)) =>
            setPosition(x, position y)
            return y
        n<0 => return insertInCache(first(cache(),(i-1)::N),scan,x,pos)
    scan := rest scan
    pos := position y
    setPosition(x, pos + DIFF)
    cach.cche := concat(cache(), x)
    x

insertInCache(before, after, x, pos) ==
    if ((pos+1) = position first after) then shiftCache(after, DIFF)
    setPosition(x, pos + ((position first after) - pos)::N quo 2))
    cach.cche := concat(before, concat(x, after))
    x

package SORTPAK SortPackage

SortPackage examples
This package exports sorting algorithms
See Also:
  o )show SortPackage
SortPackage (SORTPAK)

Exports:
  bubbleSort!  insertionSort!

-- package SORTPAK SortPackage --

)abbrev package SORTPAK SortPackage
++ Description:
++ This package exports sorting algorithms

SortPackage(S,A) : Exports == Implementation where
  S: Type
  A: IndexedAggregate(Integer,S)
    with (finiteAggregate; shallowlyMutable)

Exports == with
  bubbleSort_!: (A,(S,S) -> Boolean) -> A
    ++ bubbleSort!(a,f) undocumented
  insertionSort_!: (A, (S,S) -> Boolean) -> A
    ++ insertionSort!(a,f) undocumented
  if S has OrderedSet then
    bubbleSort_:! A -> A
    ++ bubbleSort!(a) undocumented
    insertionSort_:! A -> A
    ++ insertionSort! undocumented

Implementation == add
  bubbleSort_!(m,f) ==
    n := #m
    for i in 1..(n-1) repeat
      for j in n..(i+1) by -1 repeat
        if f(m.j,m.(j-1)) then swap_!(m,j,j-1)
      m
  insertionSort_!(m,f) ==
    for i in 2..#m repeat
      j := i
while \( j > 1 \) and \( f(m.j,m.(j-1)) \) repeat
\[
\text{swap}_!(m,j,j-1) \\
\text{\( j := (j - 1) \) pretend PositiveInteger}
\]

if \( S \) has OrderedSet then
\[
\text{bubbleSort}_!(m) == \text{bubbleSort}_!(m, _<S) \\
\text{insertionSort}_!(m) == \text{insertionSort}_!(m, _<S)
\]

if \( A \) has UnaryRecursiveAggregate\((S)\) then
\[
\text{bubbleSort}_!(m, fn) == \\
\text{\( empty? m \Rightarrow m \) \( \) \( l := m \) \( \) \( \) while not empty? \((r := l \text{\rest})\) repeat \( \text{\( r := \text{bubbleSort}_!(r, fn) \) \( x := l \text{\first} \) \( \text{\( if \text{fn}(r \text{\first}, x) \text{\then} \) \( l \text{\first} := r \text{\first} \) \( r \text{\first} := x \) \( l \text{\rest} := r \) \( l := l \text{\rest} \) \)
\]

— SORTPAK.dotabb —

"SORTPAK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SORTPAK"]
"IXAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IXAGG"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"SORTPAK" -> "IXAGG"
"SORTPAK" -> "EVALAB"

package SUP2 SparseUnivariatePolynomialFunctions2

— SparseUnivariatePolynomialFunctions2.input —

)set break resume
)sys rm -f SparseUnivariatePolynomialFunctions2.output
)spool SparseUnivariatePolynomialFunctions2.output
)set message test on
)set message auto off
)clear all
This package lifts a mapping from coefficient rings $R$ to $S$ to a mapping from sparse univariate polynomial over $R$ to a sparse univariate polynomial over $S$. Note that the mapping is assumed to send zero to zero, since it will only be applied to the non-zero coefficients of the polynomial.

See Also:
- `)show SparseUnivariatePolynomialFunctions2`
\textbf{CHAPTER 20. \textit{CHAPTER S}}

\begin{verbatim}
)abbrev package SUP2 SparseUnivariatePolynomialFunctions2
++ Description:
++ This package lifts a mapping from coefficient rings \( R \) to \( S \) to
++ a mapping from sparse univariate polynomial over \( R \) to
++ a sparse univariate polynomial over \( S \).
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

SparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
  map:(R->S,SparseUnivariatePolynomial R) -> SparseUnivariatePolynomial S
++ map(func, poly) creates a new polynomial by applying func to
++ every non-zero coefficient of the polynomial poly.
  == add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
    SparseUnivariatePolynomial R, S, SparseUnivariatePolynomial S)

— SUP2.dotabb —

"SUP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"SUP2" -> "LMODULE"
"SUP2" -> "SGROUP"

— SpecialOutputPackage.input —

)set break resume
)sys rm -f SpecialOutputPackage.output
)spool SpecialOutputPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SpecialOutputPackage
--E 1
\end{verbatim}
SpecialOutputPackage helps FORTRAN, Tex and Script Formula Formatter output from programs.

See Also:
o )show SpecialOutputPackage

Exports:
outputAsTex  outputAsFortran  outputAsScript

---

)abbrev package SPECOUT SpecialOutputPackage
++ Author: Stephen M. Watt
++ Date Created: September 1986
++ Date Last Updated: May 23, 1991
++ Description:
++ SpecialOutputPackage allows FORTRAN, Tex and
++ Script Formula Formatter output from programs.

SpecialOutputPackage: public == private where
  public == with
    outputAsFortran: (String,OutputForm) -> Void
      ++ outputAsFortran(v,o) sends output v = o in FORTRAN format
      ++ to the destination defined by \spadsyscom{set output fortran}.
    outputAsFortran: OutputForm -> Void
      ++ outputAsFortran(o) sends output o in FORTRAN format.
    outputAsScript: OutputForm -> Void
      ++ outputAsScript(o) sends output o in Script Formula Formatter format
      ++ to the destination defined by \spadsyscom{set output formula}.
    outputAsTex: OutputForm -> Void
      ++ outputAsTex(o) sends output o in Tex format to the destination
      ++ defined by \spadsyscom{set output tex}.
    outputAsFortran: List OutputForm -> Void
      ++ outputAsFortran(l) sends (for each expression in the list l)
      ++ output in FORTRAN format to the destination defined by
      ++ \spadsyscom{set output fortran}.
    outputAsScript: List OutputForm -> Void
      ++ outputAsScript(l) sends (for each expression in the list l)
      ++ output in Script Formula Formatter format to the destination defined.
      ++ by \spadsyscom{set output formula}.
    outputAsTex: List OutputForm -> Void
      ++ outputAsTex(l) sends (for each expression in the list l)
      ++ output in Tex format to the destination as defined by
      ++ \spadsyscom{set output tex}.

private == add
  e : OutputForm
  l : List OutputForm
  var : String
--ExpressionPackage()

  juxtaposeTerms: List OutputForm -> OutputForm
  juxtaposeTerms l == blankSeparate l

outputAsFortran e ==
  dispfortexp$Lisp e
  void()$Void

outputAsFortran(var,e) ==
  e := var::Symbol::OutputForm = e
  dispfortexp(e)$Lisp
  void()$Void

outputAsFortran l ==
  dispfortexp$Lisp juxtaposeTerms l
  void()$Void
package SFQCMPK SquareFreeQuasiComponentPackage

— SquareFreeQuasiComponentPackage.input —

)set break resume
)sys rm -f SquareFreeQuasiComponentPackage.output
)spool SquareFreeQuasiComponentPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SquareFreeQuasiComponentPackage
--E 1

)spool
)lisp (bye)
SquareFreeQuasiComponentPackage (SFQCMPK)

Exports:
- algebraicSort
- branchIfCan
- infRittWu?
- internalInfRittWu?
- internalSubPolSet?
- internalSubQuasiComponent?
- moreAlgebraic?
- prepareDecompose
- removeSuperfluousCases
- removeSuperfluousQuasiComponents
- startTable!
- stopTable!
- subCase?
- subPolSet?
- subQuasiComponent?
- subTriSet?
- supDimElseRittWu?

"SFQCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFQCMPK"]
package SFQCMPK SquareFreeQuasiComponentPackage

)abbrev package SFQCMPK SquareFreeQuasiComponentPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ References :
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ Tech. Report (PoSSo project)
++ d'extensions simples et resolution des systemes d'équations
++ Description:
++ A internal package for removing redundant quasi-components and redundant
++ branches when decomposing a variety by means of quasi-components
++ of regular triangular sets.

SquareFreeQuasiComponentPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
Key ==> Record(left:TS, right:TS)
Entry ==> Boolean
H ==> TabulatedComputationPackage(Key, Entry)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)
Exports == with
  startTable!: (S,S,S) -> Void
    + axiom{startTableGcd!(s1,s2,s3)}
    ++ is an internal subroutine, exported only for development.
  stopTable!: () -> Void
    + axiom{stopTableGcd!()}
    ++ is an internal subroutine, exported only for development.
  supDimElseRittWu?: (TS,TS) -> Boolean
    ++ axiom{supDimElseRittWu(ts,us)} returns true iff \text{ts}
    ++ has less elements than \text{us} otherwise if \text{ts}
    ++ has higher rank than \text{us} w.r.t. Ritt and Wu ordering.
  algebraicSort: Split -> Split
    ++ axiom{algebraicSort(lts)} sorts \text{lts} w.r.t
  moreAlgebraic?: (TS,TS) -> Boolean
    ++ axiom{moreAlgebraic?(ts,us)} returns false iff \text{ts}
    ++ and \text{us} are both empty, or \text{ts}
    ++ has less elements than \text{us}, or some variable is
    ++ algebraic w.r.t. \text{us} and is not w.r.t. \text{ts}.
  subTriSet?: (TS,TS) -> Boolean
    ++ axiom{subTriSet?(ts,us)} returns true iff \text{ts} is
    ++ a sub-set of \text{us}.
  subPolSet?: (LP,LP) -> Boolean
    ++ axiom{subPolSet?(lp1,lp2)} returns true iff \text{lp1}
    ++ is a sub-set of \text{lp2}.
  internalSubPolSet?: (LP,LP) -> Boolean
    ++ axiom{internalSubPolSet?(lp1,lp2)} returns true iff \text{lp1}
    ++ is a sub-set of \text{lp2} assuming that these lists are sorted
    ++ increasingly w.r.t.
  infRittWu?: (LP,LP) -> Boolean
    ++ axiom{infRittWu?(lp1,lp2)}
  subQuasiComponent?: (TS,TS) -> Boolean
    ++ axiom{subQuasiComponent?(ts,us)} returns true iff
    ++ internalSubQuasiComponent?(ts,us) from QuasiComponentPackage
    ++ returns true.
  subQuasiComponent?: (TS,Split) -> Boolean
    ++ axiom{subQuasiComponent?(ts,lus)} returns true iff
    ++ \text{us} in \text{lus}.\n
removeSuperfluousQuasiComponents: Split -> Split
++ \axiom{removeSuperfluousQuasiComponents(lts)} removes from
++ \axiom{lts} any \spad{ts} such that
++ \axiom{subQuasiComponent?(ts,us)} holds for
++ another \spad{us} in \axiom{lts}.
subCase?: (LpWT,LpWT) -> Boolean
++ \axiom{subCase?(lpwt1,lpwt2)}
++ is an internal subroutine, exported only for developement.
removeSuperfluousCases: List LpWT -> List LpWT
++ \axiom{removeSuperfluousCases(llpwt)}
++ is an internal subroutine, exported only for developement.
prepareDecompose: (LP, List(TS),B,B) -> List Branch
++ \axiom{prepareDecompose(lp,lts,b1,b2)}
++ is an internal subroutine, exported only for developement.
branchIfCan: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
++ \axiom{branchIfCan(leq,ts,lineq,b1,b2,b3,b4,b5)}
++ is an internal subroutine, exported only for developement.

Implementation == add

squareFreeFactors(lp: LP): LP ==
  lsflp: LP := []
  for p in lp repeat
    lsfp := squareFreeFactors(p)$polsetpack
    lsflp := concat(lsfp,lsflp)
  sort(infRittWu?,removeDuplicates lsflp)

startTable!(ok: S, ko: S, domainName: S): Void ==
  initTable!()$H
  if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
  if (not empty? domainName) then startStats!(domainName)$H
  void()

stopTable!(): Void ==
  if makingStats?()$H then printStats!()$H
  clearTable!()$H

supDimElseRittWu? (ts:TS,us:TS): Boolean ==
  #ts < #us => true
  #ts > #us => false
  lp1 :LP := members(ts)
  lp2 :LP := members(us)
  while (not empty? lp1) and (not infRittWu?(first(lp2),first(lp1)))
    repeat
      lp1 := rest lp1
      lp2 := rest lp2
    not empty? lp1

algebraicSort (lts:Split): Split ==
  lts := removeDuplicates lts
sort(supDimElseRittWu?, lts)

moreAlgebraic?(ts:TS, us:TS): Boolean ==
  empty? ts => empty? us
  empty? us => true
  #ts < #us => false
  for p in (members us) repeat
    not algebraic?(mvar(p), ts) => return false
  true

subTriSet?(ts:TS, us:TS): Boolean ==
  empty? ts => true
  empty? us => false
  mvar(ts) > mvar(us) => false
  mvar(ts) < mvar(us) => subTriSet?(ts, rest(us)::TS)
  first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS, rest(us)::TS)
  false

internalSubPolSet?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => true
  empty? lp2 => false
  associates?(first lp1, first lp2) =>
    internalSubPolSet?(rest lp1, rest lp2)
  infRittWu?(first lp1, first lp2) => false
  internalSubPolSet?(lp1, lp2)

subPolSet?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1, lp2)

infRittWu?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1, lp2)

internalInfRittWu?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?(first lp1, first lp2)$P => true
  infRittWu?(first lp2, first lp1)$P => false
  infRittWu?(rest lp1, rest lp2)$$

subCase?(lpwt1:LpWT, lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false
  subQuasiComponent?(lpwt1.tower, lpwt2.tower)

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P) then
internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
    subTriSet?(us,ts) => true
    not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
    for p in (members us) repeat
        mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
            return("failed":Union(Boolean,"failed"))
    for p in (members us) repeat
        not zero? initiallyReduce(p,ts) =>
            return("failed":Union(Boolean,"failed"))
    lsfp := squareFreeFactors(initials us)
    for p in lsfp repeat
        b: B := invertible?(p,ts)$TS
        not b =>
            return(false::Union(Boolean,"failed"))
    true::Union(Boolean,"failed")
    else

    internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
        subTriSet?(us,ts) => true
        not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
        for p in (members us) repeat
            mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
                return("failed":Union(Boolean,"failed"))
        for p in (members us) repeat
            not zero? reduceByQuasiMonic(p,ts) =>
                return("failed":Union(Boolean,"failed"))
        true::Union(Boolean,"failed")

subQuasiComponent?(ts:TS,us:TS): Boolean ==
    k: Key := [ts, us]
    e := extractIfCan(k)$H
    e case Entry => e::Entry
    ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts,us)
    b: Boolean := (ubf case Boolean) and (ubf::Boolean)
    insert!(k,b)$H
    b

subQuasiComponent?(ts:TS,lus:Split): Boolean ==
    for us in lus repeat
        subQuasiComponent?(ts,us)$B => return true
    false

removeSuperfluousCases (cases:List LpWT) ==
    #cases < 2 => cases
toSee :=
    sort((x:LpWT,y:LpWT):Boolean +-> supDimElseRittWu?(x.tower,y.tower),
        cases)
lpwt1,lpwt2 : LpWT
toSave, headmaxcases, maxcases, copymaxcases : List LpWT
while not empty? toSee repeat
    lpwt1 := first toSee
    toSee := rest toSee
    toSave := []
    for lpwt2 in toSee repeat
        if subCase?(lpwt1, lpwt2)
            then
                lpwt1 := lpwt2
            else
                if not subCase?(lpwt2, lpwt1)
                    then
                        toSave := cons(lpwt2, toSave)
        if empty? maxcases
            then
                headmaxcases := [lpwt1]
                maxcases := headmaxcases
            else
                copymaxcases := maxcases
                while (not empty? copymaxcases) and
                   (not subCase?(lpwt1, first(copymaxcases))) repeat
                    copymaxcases := rest copymaxcases
                if empty? copymaxcases
                    then
                        setrest!(headmaxcases, [lpwt1])
                headmaxcases := rest headmaxcases
    toSee := reverse toSave
maxcases

removeSuperfluousQuasiComponents(lts: Split): Split ==
lts := removeDuplicates lts
#lts < 2 => lts
toSee := algebraicSort lts
toSave, headmaxlts, maxlts, copymaxlts : Split
while not empty? toSee repeat
    ts := first toSee
    toSee := rest toSee
    toSave := []
    for us in toSee repeat
        if subQuasiComponent?(ts, us) @ B
            then
                ts := us
        else
            if not subQuasiComponent?(us, ts) @ B
                then
                    toSave := cons(us, toSave)
    if empty? maxlts
        then
            headmaxlts := [ts]
    maxlts := headmaxlts
else
    copymaxlts := maxlts
    while (not empty? copymaxlts) and _
    (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
        copymaxlts := rest copymaxlts
    if empty? copymaxlts
        then
            setrest!(headmaxlts,[ts])
            headmaxlts := rest headmaxlts
    toSee := reverse toSave
    algebraicSort maxlts

removeAssociates (lp:LP):LP ==
    removeDuplicates [primitivePart(p) for p in lp]

branchIfCan(leq: LP, ts: TS, lineq: LP, b1:B, b2:B, b3:B, b4:B, b5:B):UBF ==
    -- ASSUME pols in leq are squarefree and mainly primitive
    -- if b1 then CLEAN UP leq
    -- if b2 then CLEAN UP lineq
    -- if b3 then SEARCH for ZERO in lineq with leq
    -- if b4 then SEARCH for ZERO in lineq with ts
    -- if b5 then SEARCH for ONE in leq with lineq
    if b1
        then
            leq := removeAssociates(leq)
            leq := remove(zero?,leq)
            any?(ground?,leq) =>
                return("failed"::Union(Branch,"failed"))
    if b2
        then
            any?(zero?,lineq) =>
                return("failed"::Union(Branch,"failed"))
            lineq := removeRedundantFactors(lineq)$polsetpack
    if b3
        then
            ps: PS := construct(leq)$PS
            for q in lineq repeat
                zero? remainder(q,ps).polnum =>
                    return("failed"::Union(Branch,"failed"))
            (empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch)::UBF
    if b4
        then
            for q in lineq repeat
                zero? initiallyReduce(q,ts) =>
                    return("failed"::Union(Branch,"failed"))
    if b5
        then
            newleq: LP := []
            for p in leq repeat
                for q in lineq repeat
if \( mvar(p) = mvar(q) \)
then
\[
g := \gcd(p, q)
\]
newp := \((p \text{ exquo } g)::P\)
ground? newp =>
\[
\text{return("failed"::Union(Branch,"failed"))}
\]
newleq := cons(newp,newleq)
else
newleq := cons(p,newleq)
\[
leq := \text{newleq}
\]
leq := sort(infRittWu?, removeDuplicates leq)
((\[leq, ts, lineq\]$\text{Branch})::\text{UBF})

prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
-- if \( b1 \) then REMOVE REDUNDANT COMPONENTS in \( \text{lts} \)
-- if \( b2 \) then SPLIT the input system with \text{squareFree}
\[
\text{lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))}
\]
any?(ground?,lp) => []
empty? lts => []
if \( b1 \) then lts := removeSuperfluousQuasiComponents lts
not b2 =>
\[
[[\text{lp,ts,squareFreeFactors(initials ts)}]$\text{Branch for ts in lts}]
\]
toSee: List Branch
\[
\text{lq: LP := []}
\]
toSee := [[\text{lq,ts,squareFreeFactors(initials ts)}]$\text{Branch for ts in lts}]
empty? \( \text{lq} \) => toSee
for \( p \) in \( \text{lp} \) repeat
\[
\text{lsfp := squareFreeFactors(p)$\text{polsetpack}}
\]
branches: List Branch := []
\[
\text{lq := []}
\]
for \( f \) in \( \text{lsfp} \) repeat
for \( \text{branch in toSee repeat} \)
\[
\text{leq : LP := branch.eq}
\]
\[
\text{ts := branch.tower}
\]
\[
\text{lineq : LP := branch.ineq}
\]
ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true,true)@UBF
ubf1 case "failed" => "leave"
ubf2: UBF :=
\[
\text{branchIfCan([f],ts,lineq,false,false,true,true,true)@UBF}
\]
ubf2 case "failed" => "leave"
\[
\text{leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))}
\]
\[
\text{lineq := sort(infRittWu?,concat(ubf1.ineq,ubf2.ineq))}
\]
\[
\text{newBranch := branchIfCan(leq,ts,lineq,false,false,false,false,false)}
\]
\[
\text{branches:= cons(newBranch::Branch,branches)}
\]
\[
\text{lq := cons(f,lq)}
\]
toSee := branches
\[
\text{sort((x,y) +-> supDimElseRittWu?(x.tower,y.tower),toSee)}
\]
package SRDCMPK SquareFreeRegularSetDecompositionPackage

-- SquareFreeRegularSetDecompositionPackage.input --

)set break resume
)sys rm -f SquareFreeRegularSetDecompositionPackage.output
)spool SquareFreeRegularSetDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SquareFreeRegularSetDecompositionPackage
--E 1

)spool
)lisp (bye)

-- SquareFreeRegularSetDecompositionPackage.help --

====================================================================
SquareFreeRegularSetDecompositionPackage examples
====================================================================

A package providing a new algorithm for solving polynomial systems
by means of regular chains. Two ways of solving are provided:
in the sense of Zariski closure (like in Kalkbrener's algorithm)
or in the sense of the regular zeros (like in Wu, Wang or Lazard-
Moreno methods). This algorithm is valid for any type
of regular set. It does not care about the way a polynomial is
added in an regular set, or how two quasi-components are compared
(by an inclusion-test), or how the invertibility test is made in the tower of simple extensions associated with a regular set.

These operations are realized respectively by the domain TS and the packages QCMPPK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS).

The same way it does not care about the way univariate polynomial gcds (with coefficients in the tower of simple extensions associated with a regular set) are computed. The only requirement is that these gcds need to have invertible initials (normalized or not).

WARNING. There is no need for a user to call directly any operation of this package since they can be accessed by the domain \texttt{\textbackslash axiomType\{TS\}}. Thus, the operations of this package are not documented.

See Also:
- \texttt{\textbackslash show SquareFreeRegularSetDecompositionPackage}

---

SquareFreeRegularSetDecompositionPackage (SRDCMPK)

Exports:
- algebraicDecompose
- convert
- decompose
- internalDecompose
- KrullNumber
- numberOfVariables
- printInfo
- transcendentalDecompose
- upDateBranches

---

)abbrev package SRDCMPK SquareFreeRegularSetDecompositionPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ References:
++ Description:
++ A package providing a new algorithm for solving polynomial systems
++ by means of regular chains. Two ways of solving are provided:
++ in the sense of Zariski closure (like in Kalkbrener’s algorithm)
++ or in the sense of the regular zeros (like in Wu, Wang or Lazard-
++ Moreno methods). This algorithm is valid for any type
++ of regular set. It does not care about the way a polynomial is
++ added in an regular set, or how two quasi-components are compared
++ (by an inclusion-test), or how the invertibility test is made in
++ the tower of simple extensions associated with a regular set.
++ These operations are realized respectively by the domain \spad{TS}
++ and the packages \spad{QCMPPK(R,E,V,P,TS)} and \spad{RSETGCD(R,E,V,P,TS)}.
++ The same way it does not care about the way univariate polynomial
++ gcdd (with coefficients in the tower of simple extensions associated
++ with a regular set) are computed. The only requirement is that these
++ gcdd need to have invertible initials (normalized or not).
++ WARNING. There is no need for a user to call directly any operation
++ of this package since they can be accessed by the domain \axiomType{TS}.
++ Thus, the operations of this package are not documented.

SquareFreeRegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LPWT ==> Record(val : (List P), tower : TS)
Wip ==> Record(done: Split, todo: List LPWT)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

KrullNumber: (LP, Split) -> N
numberOfVariables: (LP, Split) -> N
algebraicDecompose: (P, TS) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P, TS, N) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P, TS) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P, TS, N, B) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P, TS, N) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P, TS) -> Record(done: Split, todo: List LpWT)
decompose: (LP, Split, B, B) -> Split
decompose: (LP, Split, B, B, B, B, B) -> Split
upDateBranches: (LP, Split, List LpWT, Wip, N) -> List LpWT
convert: Record(val: List P, tower: TS) -> String
printInfo: (List Record(val: List P, tower: TS), N) -> Void

Implementation == add

KrullNumber(lp: LP, lts: Split): N ==
ln: List N := [#(ts) for ts in lts]
n := #lp + reduce(max, ln)

numberOfVariables(lp: LP, lts: Split): N ==
lv: List V := variables([lp]$PS)
for ts in lts repeat lv := concat(variables(ts), lv)
# removeDuplicates(lv)

ground? p =>
  error " in algebraicDecompose$REGSET: should never happen !"
v := mvar(p); n := #ts
ts_v_- := collectUnder(ts, v)
ts_v_+ := collectUpper(ts, v)
ts_v := select(ts, v)::P
lgwt: List PWT
if mdeg(p) < mdeg(ts_v)
  then
    lgwt := stoseInternalLastSubResultant(ts_v, p, ts_v_, true, false)$regsetgcdpack
  else
    lgwt := stoseInternalLastSubResultant(p, ts_v, ts_v, true, false)$regsetgcdpack
lts: Split := []
llpwt: List LpWT := []
for gwt in lgwt repeat
  g := gwt.val; us := gwt.tower
  zero? g =>
  error " in algebraicDecompose$REGSET: should never happen !"
ground? g => "leave"
h := leadingCoefficient(g, v)
lus := augment(members(ts_v_+), augment(ts_v, us)$TS)$TS
lsfp := squareFreeFactors(h)$polsetpack
for f in lsfp repeat
  ground? f => "leave"
for vs in lus repeat
ltp := cons([[f,p],vs]$LpWT, ltp)

n < #us =>
  error " in algebraicDecompose$REGSET: should never happen !!!"

mvar(g) = v =>
  lts := concat(augment(members(ts_v_+),augment(g,us)$TS)$TS,lts)
[ lts, ltp ]

lts: Split
if #ts < bound then
  lts := augment(p,ts)$TS
else
  lts := []
ltp: List LpWT := []
[ lts, ltp ]

lts: Split:= augment(p,ts)$TS
ltp: List LpWT := []
[ lts, ltp ]

clos? => internalDecompose(p,ts,bound)
internalDecompose(p,ts)

  -- ASSUME p not constant
  ltp: List LpWT := []
lts: Split := []
  -- EITHER mvar(p) is null
  if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p))) then
    ltp := cons([[mvar(p)::P],ts]$LpWT,ltp)
    p := (p exquo lmp)::P
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
  -- OR init(p) is null or not
  lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
  for bwt in lbwt repeat
    bwt.val =>
      if algebraic?(mvar(p),bwt.tower)
        then
          rsl := algebraicDecompose(p,bwt.tower)
        else
          rsl := transcendentalDecompose(p,bwt.tower,bound)
        lts := concat(rsl.done,lts)
ltp := concat(rsl.todo,ltp)
    (not ground? ip) =>
      zero? tp => ltp := cons([[ip],bwt.tower]$LpWT, ltp)
(not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)

riv := removeZero(ip,bwt.tower)

(zero? riv) =>
  zero? tp => lts := cons(bwt.tower,lts)
  (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
  llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)

[lts,llpwt]

  -- ASSUME p not constant
  llpwt: List LpWT := []
  lts: Split := []
  -- EITHER mvar(p) is null
  if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p))) then
    llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
    -- OR init(p) is null or not
    lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
    for bwt in lbwt repeat
      bwt.val =>
        if algebraic?(mvar(p),bwt.tower)
          then
            rsl := algebraicDecompose(p,bwt.tower)
          else
            rsl := transcendentalDecompose(p,bwt.tower)
        lts := concat(rsl.done,lts)
        llpwt := concat(rsl.todo,llpwt)
  (not ground? ip) =>
    zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
    (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
  llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)

[llpwt]

  decompose(lp,lts,false,false,clos?,true,info?)

convert(lpwt: LpWT): String ==
  ls: List String := ["<", string((#(lpwt.val))::Z), ",", string((#(lpwt.tower))::Z), ">" ]
  concat ls

printInfo(toSee: List LpWT, n: N): Void ==
  lpwt := first toSee
  s: String := concat ["[", string((#toSee)::Z), ", ", convert(lpwt)@String]
m: N := #(lpwt.val)
toSee := rest toSee
for lpwt in toSee repeat
  m := m + #(lpwt.val)
  s := concat [s, ",", convert(lpwt)@String]
  s := concat [s, " -> |", string(m::Z), "|; {", string(n::Z),"}]]
iprint(s)$iprintpack
void()
  -- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
  -- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
  -- if clos? then SOLVE in the closure sense
  -- if rem? then REDUCE the current p by using remainder
  -- if info? then PRINT info
  empty? lp => lts
  branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$quasicomppack
  empty? branches => []
toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
toSave: Split := []
  if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
  while (not empty? toSee) repeat
    if info? then printInfo(toSee,#toSave)
    lpwt := first toSee; toSee := rest toSee
    lp := lpwt.val; ts := lpwt.tower
    empty? lp =>
      toSave := cons(ts, toSave)
    p := first lp; lp := rest lp
    if rem? and (not ground? p) and (not empty? ts)
      then
        p := remainder(p,ts).polnum
        p := removeZero(p,ts)
        zero? p => toSee := cons([lp,ts]$LpWT, toSee)
        ground? p => "leave"
        rsl := internalDecompose(p,ts,bound,clos?)
        toSee := upDateBranches(lp,toSave,toSee,rsl,bound)
removeSuperfluousQuasiComponents(toSave)$quasicomppack
  newBranches: List LpWT := wip.todo
  newComponents: Split := wip.done
  branches1, branches2: List LpWT
  branches1 := []; branches2 := []
  for branch in newBranches repeat
    us := branch.tower
    #us > n => "leave"
    newleq := sort(infRittWu?,concat(leq,branch.val))
    --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
    --any?(ground?,foo) => "leave"
    branches1 := cons([newleq,us]$LpWT, branches1)
for us in newComponents repeat

#us > n => "leave"

subQuasiComponent?(us,lts)$quasicomppack => "leave"
--newleq := leq
--foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
--any?(ground?,foo) => "leave"
branches2 := cons([leq,us]$LpWT, branches2)
empty? branches1 =>
    empty? branches2 => current
    concat(branches2, current)
branches := concat [branches2, branches1, current]
-- branches := concat(branches, current)
removeSuperfluousCases(branches)$quasicomppack

— SRDCMPK.dotabb —

"SRDCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SRDCMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SRDCMPK" -> "SFRTCAT"

package SFRGCD SquareFreeRegularTriangularSetGcd-Package

— SquareFreeRegularTriangularSetGcdPackage.input —

)set break resume
)sys rm -f SquareFreeRegularTriangularSetGcdPackage.output
)spool SquareFreeRegularTriangularSetGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SquareFreeRegularTriangularSetGcdPackage
--E 1

)spool
)lisp (bye)

—
A internal package for computing gcds and resultants of univariate polynomials with coefficients in a tower of simple extensions of a field. There is no need to use directly this package since its main operations are available from TS.

See Also:
o )show SquareFreeRegularTriangularSetGcdPackage

— package SFRGCD SquareFreeRegularTriangularSetGcdPackage —

)abbrev package SFRGCD SquareFreeRegularTriangularSetGcdPackage
CHAPTER 20. CHAPTER S

++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 10/01/1998
++ References:
++ [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++ algebraic towers of simple extensions" In proceedings of AAECC11
++ d'extensions simples et resolution des systemes d'equations
++ Description:
++ A internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.
++ There is no need to use directly this package since its main operations are
++ available from \verb|^spad|(|TS|).


\begin{verbatim}
R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
EnterGcd ==> List PWT
HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
KeyInvSet ==> Record(arg1: P, arg3: TS)
EnterInvSet ==> List TS
HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)

SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)
\end{verbatim}
Exports == with
  startTableGcd!: (S,S,S) -> Void
  stopTableGcd!: () -> Void
  startTableInvSet!: (S,S,S) -> Void
  stopTableInvSet!: () -> Void
  stosePrepareSubResAlgo: (P,P,TS) -> List LpWT
  stoseInternalLastSubResultant: (P,P,TS,B,B) -> List PWT
  stoseIntegralLastSubResultant: (P,P,TS) -> List PWT
  stoseLastSubResultant: (P,P,TS) -> List PWT
  stoseInvertible?: (P,TS) -> B
  stoseInvertible_setSqfReg: (P,TS) -> List BWT
  stoseInvertible_setReg: (P,TS) -> List BWT
  stoseInvertible?: (P,TS) -> List BWT
  stoseInvertibleSet: (P,TS) -> Split
  stoseSquareFreePart: (P,TS) -> List PWT

Implementation == add

startTableGcd!(ok: S, ko: S, domainName: S): Void ==
  initTable!()$HGcd
  printInfo!(ok,ko)$HGcd
  startStats!(domainName)$HGcd
  void()

stopTableGcd!(): Void ==
  if makingStats?()$HGcd then printStats!()$HGcd
  clearTable!()$HGcd

startTableInvSet!(ok: S, ko: S, domainName: S): Void ==
  initTable!()$HInvSet
  printInfo!(ok,ko)$HInvSet
  startStats!(domainName)$HInvSet
  void()

stopTableInvSet!(): Void ==
  if makingStats?()$HInvSet then printStats!()$HInvSet
  clearTable!()$HInvSet

  q := primitivePart initiallyReduce(p,ts)
  zero? q => false
  normalized?(q,ts) => true
  v := mvar(q)
  not algebraic?(v,ts) =>
    toCheck: List BWT := stoseInvertible?(p,ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val = false => return false
CHAPTER 20. CHAPTER S

return true

\[ts_v := \text{select}(ts,v)\]::P

\[ts_v_ - := \text{collectUnder}(ts,v)\]

\[lgwt := \text{stoseInternalLastSubResultant}(ts_v,q,ts_v_ -,false,true)\]

for gwt in lgwt repeat

\[g := gwt.val;\]

\[(\text{not ground? } g) \text{ and } (\text{mvar}(g) = v) =>\]

return false

true


-- ASSUME \text{mvar}(p1) = \text{mvar}(p2) > \text{mvar}(ts) \text{ and } \text{mdeg}(p1) >= \text{mdeg}(p2)
-- ASSUME \text{init}(p1) invertible modulo \text{ts} !!!

toSee: List LpWT := [[[p1,p2],ts]$LpWT]
toSave: List LpWT := []

\[v := \text{mvar}(p1)\]

while (not empty? toSee) repeat

\[lpwt := \text{first toSee}; toSee := \text{rest toSee}\]

\[p1 := \text{lpwt.val.1}; p2 := \text{lpwt.val.2}\]

\[ts := \text{lpwt.tower}\]

\[lbwt := \text{stoseInvertible?}(\text{leadingCoefficient}(p2,v),ts)@(\text{List BWT})\]

for bwt in lbwt repeat

\[(\text{bwt.val} = \text{true}) \text{ and } (\text{degree}(p2,v) > 0) =>\]

\[p3 := \text{prem}(p1, -p2)\]

\[s: P := \text{init}(p2)**(\text{mdeg}(p1) - \text{mdeg}(p2))::N\]

\[\text{toSave} := \text{cons}([[p2,p3,s],bwt.tower]$LpWT,\text{toSave})\]

-- \text{p2} := \text{initiallyReduce}(p2,bwt.tower)

newp2 := \text{primitivePart initiallyReduce}(p2,bwt.tower)

\[(\text{bwt.val} = \text{true}) =>\]

-- \text{toSave} := \text{cons}([[p2,0,1],bwt.tower]$LpWT,\text{toSave})

\[\text{toSave} := \text{cons}([[p2,0,1],bwt.tower]$LpWT,\text{toSave})\]

-- \text{zero? } p2 =>

zero? newp2 =>

\[\text{toSave} := \text{cons}([[p1,0,1],bwt.tower]$LpWT,\text{toSave})\]

-- \text{toSee} := \text{cons}([[p1,p2],bwt.tower]$LpWT,\text{toSee})

\[\text{toSee} := \text{cons}([[p1,newp2],bwt.tower]$LpWT,\text{toSee})\]

\[\text{toSave}\]

stoseIntegralLastSubResultant\((p1:P,p2:P,ts:TS)\): List PWT ==

-- ASSUME \text{mvar}(p1) = \text{mvar}(p2) > \text{mvar}(ts) \text{ and } \text{mdeg}(p1) >= \text{mdeg}(p2)
-- ASSUME \text{p1} and \text{p2} have no algebraic coefficients

\[\text{lsr} := \text{lastSubResultant}(p1, p2)\]

\[\text{ground?}(\text{lsr}) => [[\text{lsr},ts]$PWT]\]

\[\text{mvar}(\text{lsr}) < \text{mvar}(p1) => [[\text{lsr},ts]$PWT]\]

\[\text{gii12} := \text{gcd}(\text{init}(p1),\text{init}(p2))\]

\[\text{ex: } \text{Union}(P,"\text{failed}) := (\text{gii12} * \text{lsr}) \text{ exquo } P \text{ init}(\text{lsr})\]

\[\text{ex case } \text{"failed" } => [[\text{lsr},ts]$PWT]\]

\[[\text{ex:}$P,ts]$PWT]\n
stoseInternalLastSubResultant\((p1:P,p2:P,ts:TS,b1:B,b2:B)\): List PWT ==
-- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
-- if b1 ASSUME init(p2) invertible w.r.t. ts
-- if b2 BREAK with the first non-trivial gcd
k: KeyGcd := [p1,p2,ts,b2]
e := extractIfCan(k)$HGcd
e case EntryGcd => e::EntryGcd
toSave: List PWT
empty? ts =>
toSave := stoseIntegralLastSubResultant(p1,p2,ts)
insert!(k,toSave)$HGcd
return toSave
toSee: List LpWT
if b1
then
  p3 := prem(p1, -p2)
  s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
toSee := [[p2,p3,s],ts]$LpWT
else
  toSee := stosePrepareSubResAlgo(p1,p2,ts)
toSave := stoseInternalLastSubResultant(toSee,mvar(p1),b2)
insert!(k,toSave)$HGcd
toSave

stoseInternalLastSubResultant(llpwt: List LpWT,v:V,b2:B): List PWT ==
toReturn: List PWT := []; toSee: List LpWT;
while (not empty? llpwt) repeat
toSee := llpwt; llpwt := []
-- CONSIDER FIRST the vanishing current last subresultant
for lpwt in toSee repeat
  p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
  ts := lpwt.tower
  lbwt := stoseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
  for bwt in lbwt repeat
    bwt.val = false =>
    toReturn := cons([[p1,bwt.tower]$PWT, toReturn]
    b2 and positive?(degree(p1,v)) => return toReturn
    llpwt := cons([[p1,p2,s],bwt.tower]$LpWT, llpwt)
empty? llpwt => "leave"
-- CONSIDER NOW the branches where the computations continue
toFloat := llpwt; llpwt := []
lpwt := first toSee; toSee := rest toSee
p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
delta: N := (mdeg(p1) - degree(p2,v))::N
p3: P := LazardQuotient2(p2, leadingCoefficient(p2,v), s, delta)
zero?(degree(p3,v)) =>
toReturn := cons([[p3,lpwt.tower]$PWT, toReturn]
for lpwt in toSee repeat
  toReturn := cons([[p3,lpwt.tower]$PWT, toReturn]
\[
(p_1, p_2) := (p_3, \text{next\_subResultant2}(p_1, p_2, p_3, s))
\]
\[
s := \text{leadingCoefficient}(p_1, v)
\]
\[
\text{llpwt} := \text{cons}([[p_1, p_2, s], \text{lpwt}.\text{tower}] \text{$LpWT$}, \text{llpwt})
\]
\[
\text{for} \ \text{lpwt in toSee repeat}
\]
\[
\text{llpwt} := \text{cons}([[p_1, p_2, s], \text{lpwt}.\text{tower}] \text{$LpWT$}, \text{llpwt})
\]
\[
\text{toreturn}
\]

\[
\text{stoseLastSubResultant}(p_1:P, p_2:P, ts:TS) := \text{List PWT} ==
\]
\[
\text{ground? p}_1 \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #1"}
\]
\[
\text{ground? p}_2 \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"}
\]
\[
\text{not (mvar}(p_2) = \text{mvar}(p_1)) \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"
\]
\[
\text{algebraic?}(\text{mvar}(p_1), ts) \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #1"
\]
\[
\text{not initiallyReduced?}(p_1, ts) \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #1"
\]
\[
\text{not initiallyReduced?}(p_2, ts) \Rightarrow
g\text{error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"
\]
\[
\text{purelyTranscendental?}(p_1, ts) \text{ and purelyTranscendental?}(p_2, ts) \Rightarrow
\text{stoseIntegralLastSubResultant}(p_1, p_2, ts)
\]
\[
\text{if } \text{mdeg}(p_1) < \text{mdeg}(p_2) \text{ then}
\]
\[
(p_1, p_2) := (p_2, p_1)
\]
\[
\text{if odd?}(\text{mdeg}(p_1)) \text{ and odd?}(\text{mdeg}(p_2)) \text{ then } p_2 := - p_2
\]
\[
\text{stoseInternalLastSubResultant}(p_1, p_2, ts, false, false)
\]

\[
\text{stoseSquareFreePart\_wip}(p:P, ts:TS) := \text{List PWT} ==
\]
\[
\text{-- ASSUME p is not constant and mvar(p) > mvar(ts)}
\]
\[
\text{-- ASSUME init(p) is invertible w.r.t. ts}
\]
\[
\text{-- ASSUME p is mainly primitive}
\]
\[
\text{-- one? mdeg(p) \Rightarrow [[p,ts]$PWT]}
\]
\[
\text{mdeg}(p) = 1 \Rightarrow [[p,ts]$PWT]
\]
\[
v := \text{mvar}(p)$P
\]
\[
q := \text{mainPrimitivePart D}(p, v)
\]
\[
l\text{gwt} := \text{List PWT} := \text{stoseInternalLastSubResultant}(p, q, ts, true, false)
\]
\[
lpwt := \text{List PWT} := []
\]
\[
sfp := P
\]
\[
\text{for gwt in lgwt repeat}
\]
\[
g := gwt.val; us := gwt.tower
\]
\[
(\text{ground? g}) \text{ or (mvar}(g) < v) \Rightarrow
\]
\[
lpwt := \text{cons}([[p,us],lpwt)
\]
\[
g := \text{mainPrimitivePart g}
\]
\[
sfp := \text{lazyPquo}(p, g)
\]
\[
sfp := \text{mainPrimitivePart stronglyReduce}(sfp, us)
\]
\[
lpwt := \text{cons}([[sfp,us],lpwt)
\]
\[
lpwt
\]

\[
\]

stoseInvertible?_sqfreg(p:P,ts:TS): List BWT ==
  --iprint("+")$iprintpack
  q := primitivePart initiallyReduce(p,ts)
  zero? q => [[false,ts]$BWT]
  normalized?(q,ts) => [[true,ts]$BWT]
  v := mvar(q)
  not algebraic?(v,ts) =>
    lbwt: List BWT := []
    toCheck: List BWT := stoseInvertible?_sqfreg(init(q),ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val => lbwt := cons(bwt,lbwt)
      newq := removeZero(q,bwt.tower)
      zero? newq => lbwt := cons(bwt,lbwt)
      lbwt :=
        concat(stoseInvertible?_sqfreg(newq,bwt.tower)@(List BWT), lbwt)
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,$false,false)
  lbwt: List BWT := []
  lts, lts_g, lts_h: Split
  for gwt in lgwt repeat
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
      lts := augment(ts_v,ts)$TS
      lts := augment(members(ts_v_+),lts)$TS
      for ts in lts repeat
        lbwt := cons([[true, ts]$BWT,lbwt]
    g := mainPrimitivePart g
    lts_g := augment(g,ts)$TS
    lts_g := augment(members(ts_v_+),lts_g)$TS
    -- USE stoseInternalAugment with parameters ??
    for ts_g in lts_g repeat
      lbwt := cons([[false, ts_g]$BWT,lbwt]
    h := lazyPquo(ts_v,g)
    (ground? h) or (mvar(h) < v) => "leave"
    h := mainPrimitivePart h
    lts_h := augment(h,ts)$TS
    lts_h := augment(members(ts_v_+),lts_h)$TS
    -- USE stoseInternalAugment with parameters ??
    for ts_h in lts_h repeat
      lbwt := cons([[true, ts_h]$BWT,lbwt]
  sort((x,y) +-> x.val < y.val,lbwt)

  --iprint("*")$iprintpack
  k: KeyInvSet := [p,ts]
e := extractIfCan(k)$HInvSet

\text{e case EntryInvSet \Rightarrow e::EntryInvSet}

q := \text{primitivePart initiallyReduce}(p, ts)

\text{zero? q \Rightarrow []}

\text{normalized?(q, ts) \Rightarrow [ts]}

v := \text{mvar}(q)

toSave: \text{Split} := []

\text{not algebraic?}(v, ts) \Rightarrow

\text{toCheck: List BWT := stoseInvertible?_sqfreg\text{(init}(q), ts)@(List BWT)}

\text{for bwt in toCheck repeat}

bwt.val \Rightarrow toSave := cons(bwt.tower, toSave)

\text{zero? newq \Rightarrow "leave"}

\text{toSave := concat(stoseInvertibleSet_sqfreg(newq, bwt.tower), toSave)}

\text{toSave := removeDuplicates toSave}

\text{return algebraicSort(toSave)$quasicomppack}

ts_v := \text{select}(ts, v)::P

ts_v_- := \text{collectUnder}(ts, v)

ts_v_+ := \text{collectUpper}(ts, v)

\text{lgwt := stoseInternalLastSubResultant}(ts_v, q, ts_v_-)\text{, false, false}\

lts, lts_h: \text{Split}

\text{for gwt in lgwt repeat}

g := gwt.val; ts := gwt.tower

(ground? g) or (mvar(g) < v) \Rightarrow

lts := augment(ts_v, ts)$TS

\text{lts := augment(members(ts_v_+), lts)$TS}

\text{toSave := concat(lts, toSave)}

\text{g := mainPrimitivePart g}

h := lazyPquo(ts_v, g)

\text{h := mainPrimitivePart h}

(ground? h) or (mvar(h) < v) \Rightarrow "leave"

\text{lts_h := augment(h, ts)$TS}

\text{lts_h := augment(members(ts_v_+), lts_h)$TS}

\text{toSave := concat(lts_h, toSave)}

\text{toSave := algebraicSort(toSave)$quasicomppack}

\text{insert!(k, toSave)$HInvSet}

toSave

\text{stoseInvertible?_reg(p:P, ts:TS): List BWT ==}

--iprint("-")$iprintpack

q := \text{primitivePart initiallyReduce}(p, ts)

\text{zero? q \Rightarrow [[false, ts]$BWT]}

\text{normalized?(q, ts) \Rightarrow [[true, ts]$BWT]}

v := \text{mvar}(q)

\text{not algebraic?}(v, ts) \Rightarrow

\text{lbt: List BWT := []}

\text{toCheck: List BWT := stoseInvertible?_reg\text{(init}(q), ts)@(List BWT)}

\text{for bwt in toCheck repeat}

bwt.val \Rightarrow \text{lbt := cons(bwt, lbt)}

\text{newq := removeZero(q, bwt.tower)
zero? newq => lbwt := cons(bwt,lbwt)
lbwt :=
concat(stoseInvertible?_reg(newq,bwt.tower)@(List BWT), lbwt)
return lbwt
ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
lbwt: List BWT := []
lts, lts_g, lts_h: Split
for gwt in lgwt repeat
g := gwt.val; ts := gwt.tower
(ground? g) or (mvar(g) < v) =>
lts := augment(ts_v,ts)$TS
lts := augment(members(ts_v_+),lts)$TS
for ts in lts repeat
lbwt := cons([true, ts]$BWT,lbwt)
g := mainPrimitivePart g
lts_g := augment(g,ts)$TS
lts_g := augment(members(ts_v_+),lts_g)$TS
-- USE internalAugment with parameters ??
for ts_g in lts_g repeat
lbwt := cons([false, ts_g]$BWT,lbwt)
h := lazyPquo(ts_v,g)
(ground? h) or (mvar(h) < v) => "leave"
h := mainPrimitivePart h
lts_h := augment(h,ts)$TS
lts_h := augment(members(ts_v_+),lts_h)$TS
-- USE internalAugment with parameters ??
for ts_h in lts_h repeat
inv := stoseInvertible?_reg(q,ts_h)@(List BWT)
lbwt := concat([bwt for bwt in inv | bwt.val],lbwt)
sort((x,y) +-> x.val < y.val,lbwt)

--iprint("/")$iprintpack
k: KeyInvSet := [p,ts]
e := extractIfCan(k)$HInvSet
e case EntryInvSet => e::EntryInvSet
q := primitivePart initiallyReduce(p,ts)
zero? q => []
normalized?(q,ts) => [ts]
v := mvar(q)
toSave: Split := []
not algebraic?(v,ts) =>
toCheck: List BWT := stoseInvertible?_reg(init(q),ts)@(List BWT)
for bwt in toCheck repeat
bwt.val => toSave := cons(bwt.tower,toSave)
newq := removeZero(q,bwt.tower)
zero? newq => "leave"
toSave := concat(stoseInvertibleSet_reg(newq,bwt.tower), toSave)
toSave := removeDuplicates toSave
return algebraicSort(toSave)$quasicomppack
ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
lts, lts_h: Split
for gwt in lgwt repeat
  g := gwt.val; ts := gwt.tower
  (ground? g) or (mvar(g) < v) =>
    lts := augment(ts_v,ts)$TS
    lts := augment(members(ts_v_+),lts)$TS
toSave := concat(lts,toSave)
g := mainPrimitivePart g
h := lazyPquo(ts_v,g)
h := mainPrimitivePart h
  (ground? h) or (mvar(h) < v) => "leave"
lts_h := augment(h,ts)$TS
lts_h := augment(members(ts_v_+),lts_h)$TS
for ts_h in lts_h repeat
  inv := stoseInvertibleSet_reg(q,ts_h)
toSave := removeDuplicates concat(inv,toSave)
toSave := algebraicSort(toSave)$quasicomppack
insert!(k,toSave)$HInvSet
toSave

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P)
then
else
package MATSTOR StorageEfficientMatrixOperations

---

StorageEfficientMatrixOperations.input

)set break resume
)sys rm -f StorageEfficientMatrixOperations.output
)spool StorageEfficientMatrixOperations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StorageEfficientMatrixOperations
--E 1

)spool
)lisp (bye)

---

StorageEfficientMatrixOperations.help

====================================================================
StorageEfficientMatrixOperations examples
====================================================================

This package provides standard arithmetic operations on matrices. The functions in this package store the results of computations in existing matrices, rather than creating new matrices. This package works only for matrices of type Matrix and uses the internal representation of this type.

See Also:
  o )show StorageEfficientMatrixOperations
StorageEfficientMatrixOperations (MATSTOR)

Exports:
  copy!  leftScalarTimes!  minus!  plus!  power!
  rightScalarTimes!  times!  ?**?

--- package MATSTOR StorageEfficientMatrixOperations ---

)abbrev package MATSTOR StorageEfficientMatrixOperations
++ Author: Clifton J. Williamson
++ Date Created: 18 July 1990
++ Date Last Updated: 18 July 1990
++ Description:
++ This package provides standard arithmetic operations on matrices.
++ The functions in this package store the results of computations
++ in existing matrices, rather than creating new matrices. This
++ package works only for matrices of type Matrix and uses the
++ internal representation of this type.

StorageEfficientMatrixOperations(R): Exports == Implementation where
R : Ring
M ==> Matrix R
NNI ==> NonNegativeInteger
ARR ==> PrimitiveArray R
REP ==> PrimitiveArray PrimitiveArray R

Exports ==> with
  copy_! : (M,M) -> M
  ++ \spad{copy!(c,a)} copies the matrix \spad{a} into the matrix \spad{c}.
  ++ Error: if \spad{a} and \spad{c} do not have the same
  ++ dimensions.
  plus_! : (M,M,M) -> M
  ++ \spad{plus!(c,a,b)} computes the matrix sum \spad{a + b} and stores the
  ++ result in the matrix \spad{c}.
  ++ Error: if \spad{a}, \spad{b}, and \spad{c} do not have the same dimensions.
  minus_! : (M,M) -> M
  ++ \spad{minus!(c,a)} computes \spad{-a} and stores the result in the
++ matrix c.
++ Error: if a and c do not have the same dimensions.

minus_! : (M,M,M) -> M
++ \spad{\text{\textbackslash minus!(c,a,b)}} computes the matrix difference \spad{c - b}
++ and stores the result in the matrix c.
++ Error: if \spad{\text{\textbackslash}{\ spad{a}}, b, and c do not have the same dimensions.}

leftScalarTimes_! : (M,R,M) -> M
++ \spad{\text{\textbackslash leftScalarTimes!(c,r,a)}} computes the scalar product
++ \spad{r \times a} and stores the result in the matrix c.
++ Error: if \spad{\text{\textbackslash}{\ spad{a}} and c do not have the same dimensions.}

rightScalarTimes_! : (M,M,R) -> M
++ \spad{\text{\textbackslash rightScalarTimes!(c,a,r)}} computes the scalar product
++ \spad{a \times r} and stores the result in the matrix c.
++ Error: if \spad{\text{\textbackslash}{\ spad{a}} and c do not have the same dimensions.}

times_! : (M,M,M) -> M
++ \spad{\text{\textbackslash times!(c,a,b)}} computes the matrix product \spad{a \times b}
++ and stores the result in the matrix c.
++ Error: if \spad{\text{\textbackslash}{\ spad{a}}, b, and c do not have
++ compatible dimensions.}

power_! : (M,M,M,M,NNI) -> M
++ \spad{\text{\textbackslash power!(a,b,c,m,n)}} computes m ** n and stores the result in
++ \spad{\text{\textbackslash}{\ spad{a}}}. The matrices b and c are used to store intermediate results.
++ Error: if \spad{\text{\textbackslash}{\ spad{a}}, b, c, and m are not square
++ and of the same dimensions.}

"**" : (M,NNI) -> M
++ \spad{x ** n} computes the n-th power
++ of a square matrix. The power n is assumed greater than 1.

Implementation ==> add

rep : M -> REP
rep m == m pretend REP

copy_!((c,a) ==
m := nrows a; n := ncols a
not((nrows c) = m and (ncols c) = n) =>
  error "copy!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
for i in 0..(m-1) repeat
  aRow := qelt(aa,i); cRow := qelt(cc,i)
  for j in 0..(n-1) repeat
    qsetelt_!(cRow,j,qelt(aRow,j))
c

copy_!((c,a,b) ==
m := nrows a; n := ncols a
not((nrows b) = m and (ncols b) = n) =>
  error "plus!: matrices of incompatible dimensions"
not((nrows c) = m and (ncols c) = n) =>
  error "plus!: matrices of incompatible dimensions"
aa := rep a; bb := rep b; cc := rep c
for i in 0..(m-1) repeat
    aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,qelt(aRow,j) + qelt(bRow,j))
    c

minus_!(c,a) ==
m := nrows a; n := ncols a
not((nrows c) = m and (ncols c) = n) =>
    error "minus!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,-qelt(aRow,j))
    c

minus_!(c,a,b) ==
m := nrows a; n := ncols a
not((nrows b) = m and (ncols b) = n) =>
    error "minus!: matrices of incompatible dimensions"
not((nrows c) = m and (ncols c) = n) =>
    error "minus!: matrices of incompatible dimensions"
aa := rep a; bb := rep b; cc := rep c
for i in 0..(m-1) repeat
    aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,qelt(aRow,j) - qelt(bRow,j))
    c

leftScalarTimes_!(c,r,a) ==
m := nrows a; n := ncols a
not((nrows c) = m and (ncols c) = n) =>
    error "leftScalarTimes!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,r * qelt(aRow,j))
    c

rightScalarTimes_!(c,a,r) ==
m := nrows a; n := ncols a
not((nrows c) = m and (ncols c) = n) =>
    error "rightScalarTimes!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
times_!(c,a,b) ==
    m := nrows a; n := ncols a; p := ncols b
    not((nrows b) = n and (nrows c) = m and (ncols c) = p) =>
        error "times!: matrices of incompatible dimensions"
    aa := rep a; bb := rep b; cc := rep c
    bCol : ARR := new(n,0)
    m1 := (m :: Integer) - 1; n1 := (n :: Integer) - 1
    for j in 0..(p-1) repeat
        copyCol_!(bCol,bb,j,n1)
    for i in 0..m1 repeat
        aRow := qelt(aa,i); cRow := qelt(cc,i)
        sum : R := 0
        for k in 0..n1 repeat
            sum := sum + qelt(aRow,k) * qelt(bCol,k)
            qsetelt_!(cRow,j,sum)
c
power_!(a,b,c,m,p) ==
    mm := nrows a; nn := ncols a
    not(mm = nn) =>
        error "power!: matrix must be square"
    not((nrows b) = mm and (ncols b) = nn) =>
        error "power!: matrices of incompatible dimensions"
    not((nrows c) = mm and (ncols c) = nn) =>
        error "power!: matrices of incompatible dimensions"
    not((nrows m) = mm and (ncols m) = nn) =>
        error "power!: matrices of incompatible dimensions"
    flag := false
    copy_!(b,m)
    repeat
        if odd? p then
            flag =>
                times_!(c,b,a)
                copy_!(a,c)
                flag := true
                copy_!(a,b)
        -- one? p => return a
        (p = 1) => return a
        p := p quo 2
        times_!(c,b,b)
        copy_!(b,c)
    m ** n ==

not square? m => error "**: matrix must be square"
a := copy m; b := copy m; c := copy m
power!(a,b,c,m,n)
o )show StreamFunctions1

---

**StreamFunctions1 (STREAM1)**

---

**Exports:**

concat

---

```spad
)abbrev package STREAM1 StreamFunctions1
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in one set.

StreamFunctions1(S:Type): Exports == Implementation where
  ST ==> Stream

Exports ==> with
  concat: ST ST S -> ST S
  ++ concat(u) returns the left-to-right concatenation of the
  ++ streams in u. Note that \spad{concat(u) = reduce(concat,u)}.
  ++
  ++X m:=[i for i in 10..]
  ++X n:=[j for j in 1.. | prime? j]
  ++X p:=[m,n]:Stream(Stream(PositiveInteger))
  ++X concat(p)

Implementation ==> add
```
\begin{verbatim}
concat z == delay
  empty? z => empty()
  empty?(x := frst z) => concat rst z
  concat(frst x,concat(rst x,concat rst z))

---

<table>
<thead>
<tr>
<th>STREAM1.dotabb</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;STREAM1&quot; [color=&quot;#FF4488&quot;,href=&quot;bookvol10.4.pdf#nameddest=STREAM1&quot;]</td>
</tr>
<tr>
<td>&quot;TYPE&quot; [color=&quot;#4488FF&quot;,href=&quot;bookvol10.2.pdf#nameddest=TYPE&quot;]</td>
</tr>
<tr>
<td>&quot;STREAM1&quot; -&gt; &quot;TYPE&quot;</td>
</tr>
</tbody>
</table>

---

package STREAM2 StreamFunctions2

---

<table>
<thead>
<tr>
<th>StreamFunctions2.input</th>
</tr>
</thead>
<tbody>
<tr>
<td>)set break resume</td>
</tr>
<tr>
<td>)sys rm -f StreamFunctions2.output</td>
</tr>
<tr>
<td>)spool StreamFunctions2.output</td>
</tr>
<tr>
<td>)set message test on</td>
</tr>
<tr>
<td>)set message auto off</td>
</tr>
<tr>
<td>)clear all</td>
</tr>
<tr>
<td>--S 1 of 1</td>
</tr>
<tr>
<td>)show StreamFunctions2</td>
</tr>
<tr>
<td>--E 1</td>
</tr>
<tr>
<td>)spool</td>
</tr>
<tr>
<td>)lisp (bye)</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>StreamFunctions2.help</th>
</tr>
</thead>
<tbody>
<tr>
<td>====================================================================</td>
</tr>
<tr>
<td>StreamFunctions2 examples</td>
</tr>
<tr>
<td>====================================================================</td>
</tr>
</tbody>
</table>

Functions defined on streams with entries in two sets.
See Also:
  o )show StreamFunctions2

StreamFunctions2 (STREAM2)

Exports:
  map  reduce  scan

— package STREAM2 StreamFunctions2 —

)abbrev package STREAM2 StreamFunctions2
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in two sets.

StreamFunctions2(A:Type,B:Type): Exports == Implementation where
ST   ==> Stream

Exports ==> with
  map: ((A -> B),ST A) -> ST B
    ++ map(f,s) returns a stream whose elements are the function f applied
    ++ to the corresponding elements of s.
    ++ Note that \spad{map(f,[x0,x1,x2,...]) = [f(x0),f(x1),f(x2),...]}.
    ++
    ++X m:=[i for i in 1..]
    ++X f(i:PositiveInteger):PositiveInteger==i**2
    ++X map(f,m)

  scan: (B,((A,B) -> B),ST A) -> ST B
++ scan(b,h,[x0,x1,x2,...]) returns \spad{[y0,y1,y2,...]}, where
++ \spad{y0 = h(x0,b)},
++ \spad{y1 = h(x1,y0)}, \spad{...}
++ \spad{yn = h(xn,y(n-1))}.
++

++X m:=[i for i in 1..]:Stream(Integer)
++X f(i:Integer,j:Integer):Integer==i+j
++X scan(1,f,m)

reduce: (B,(A,B) -> B,ST A) -> B
++ returns the value \spad{r(n)} computed as follows:
++ \spad{r0 = f(x0,b)},
++ \spad{r1 = f(x1,r0)}, \spad{...},
++ \spad{r(n) = f(xn,r(n-1))}.
++
++X m:=[i for i in 1..300]:Stream(Integer)
++X f(i:Integer,j:Integer):Integer==i+j
++X reduce(1,f,m)

-- rreduce: (B,(A,B) -> B,ST A) -> B
-- ++ returns the value \spad{r(n)} computed as follows:
-- \spad{r0 = f(x0,b)},
-- \spad{r1 = f(x1,r0)}, \spad{...},
-- \spad{r(n) = f(xn,r(n-1))}.
--
--X reshape(y,x) = y

Implementation => add

mapp: (A -> B,ST A) -> ST B
mapp(f,x)== delay
   empty? x => empty()
   concat(f frst x, map(f,rst x))

map(f,x) ==
   explicitlyEmpty? x => empty()
   eq?(x,rst x) => repeating([f frst x])
   mapp(f,x)

-- reshape(y,x) = y

scan(b,h,x) == delay
   empty? x => empty()
   c := h(frst x,b)
   concat(c,scan(c,h,rst x))

reduce(b,h,x) ==
   empty? x => b
   reduce(h(frst x,b),h,rst x)
-- rreduce(b,h,x) ==
--   empty? x => b
--   h(frst x,rreduce(b,h,rst x))
package STREAM3 StreamFunctions3

Functions defined on streams with entries in three sets.

See Also:
  o )show StreamFunctions3
StreamFunctions3 (STREAM3)

Exports:
map

— package STREAM3 StreamFunctions3 —

)abbrev package STREAM3 StreamFunctions3
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in three sets.

StreamFunctions3(A,B,C): Exports == Implementation where
  A : Type
  B : Type
  C : Type
  ST ==> Stream

Exports ==> with
  map: ((A,B) -> C,ST A,ST B) -> ST C
  ++ map(f,st1,st2) returns the stream whose elements are the
  ++ function f applied to the corresponding elements of st1 and st2.
  ++ \spad{map(f,[x0,x1,x2,..],[y0,y1,y2,..]) = [f(x0,y0),f(x1,y1),..]}.
  ++
  ++ S
  ++X m:=[i for i in 1..]::Stream(Integer)
  ++X n:=[i for i in 1..]::Stream(Integer)
  ++X f(i:Integer,j:Integer):Integer == i+j
  ++X map(f,m,n)

Implementation ==> add

  mapp:((A,B) -> C,ST A,ST B) -> ST C
  mapp(g,x,y) == delay
    empty? x or empty? y => empty()
concat(g(frst x, frst y), map(g, rst x, rst y))

map(g, x, y) ==
  explicitlyEmpty? x => empty()
  eq?(x, rst x) => map(z +-> g(frst x, z), y)$StreamFunctions2(B, C)
  explicitlyEmpty? y => empty()
  eq?(y, rst y) => map(z +-> g(z, frst y), x)$StreamFunctions2(A, C)
  mapp(g, x, y)

---

STREAM3.dotabb

"STREAM3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM3" -> "TYPE"

---

package STINPROD StreamInfiniteProduct

--- StreamInfiniteProduct.input ---

)set break resume
)sys rm -f StreamInfiniteProduct.output
)spool StreamInfiniteProduct.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamInfiniteProduct
--E 1

)spool
)lisp (bye)

---

--- StreamInfiniteProduct.help ---

*********************************************************************
StreamInfiniteProduct examples
This package computes infinite products of Taylor series over an integral domain of characteristic 0. Here Taylor series are represented by streams of Taylor coefficients.

See Also:
- )show StreamInfiniteProduct

StreamInfiniteProduct (STINPROD)

Exports:
- evenInfiniteProduct
- generalInfiniteProduct
- infiniteProduct
- oddInfiniteProduct

— package STINPROD StreamInfiniteProduct —

)abbrev package STINPROD StreamInfiniteProduct
++ Author: Clifton J. Williamson
++ Date Created: 23 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of Taylor series over an integral domain of characteristic 0. Here Taylor series are represented by streams of Taylor coefficients.

StreamInfiniteProduct(Coef): Exports == Implementation where
Coef: Join(IntegralDomain,CharacteristicZero)
I ==> Integer
QF ==> Fraction
ST ==> Stream
Exports ==> with

infiniteProduct: ST Coef -> ST Coef
++ infiniteProduct(f(x)) computes \( \text{product}(n=1,2,3...,f(x^n)) \).
++ The series \( \text{product}(f(x)) \) should have constant coefficient 1.
evenInfiniteProduct: ST Coef -> ST Coef
++ evenInfiniteProduct(f(x)) computes \( \text{product}(n=2,4,6...,f(x^n)) \).
++ The series \( \text{product}(f(x)) \) should have constant coefficient 1.
oddInfiniteProduct: ST Coef -> ST Coef
++ oddInfiniteProduct(f(x)) computes \( \text{product}(n=1,3,5...,f(x^n)) \).
++ The series \( \text{product}(f(x)) \) should have constant coefficient 1.
generalInfiniteProduct: (ST Coef,1,I) -> ST Coef
++ generalInfiniteProduct(f(x),a,d) computes
++ \( \text{product}(n=a,a+d,a+2*d,...,f(x^n)) \).
++ The series \( \text{product}(f(x)) \) should have constant coefficient 1.

Implementation ==> add

if Coef has Field then

import StreamTaylorSeriesOperations(Coef)
import StreamTranscendentalFunctions(Coef)

infiniteProduct st == exp lambert log st
evenInfiniteProduct st == exp evenlambert log st
oddInfiniteProduct st == exp oddlambert log st
generalInfiniteProduct(st,a,d) == exp generalLambert(log st,a,d)

else

import StreamTaylorSeriesOperations(QF Coef)
import StreamTranscendentalFunctions(QF Coef)

applyOverQF:(ST QF Coef -> ST QF Coef,ST Coef) -> ST Coef
applyOverQF(f,st) ==
  stQF := map(z1 +-> z1::QF(Coef),st)$StreamFunctions2(Coef,QF Coef)
  map(z1 +-> retract(z1)@Coef,f stQF)$StreamFunctions2(QF Coef,Coef)
infiniteProduct st == applyOverQF(z1 +-> exp lambert log z1,st)
evenInfiniteProduct st == applyOverQF(z1 +-> exp evenlambert log z1,st)
oddInfiniteProduct st == applyOverQF(z1 +-> exp oddlambert log z1,st)
generalInfiniteProduct(st,a,d) ==
  applyOverQF(z1 +-> exp generalLambert(log z1,a,d),st)
package STTAYLOR StreamTaylorSeriesOperations

-- StreamTaylorSeriesOperations.input --

set break resume
sys rm -f StreamTaylorSeriesOperations.output
spool StreamTaylorSeriesOperations.output
set message test on
set message auto off
clear all
--S 1 of 1
show StreamTaylorSeriesOperations
--E 1
spool
lisp (bye)

-- StreamTaylorSeriesOperations.help --

StreamTaylorSeriesOperations examples

StreamTaylorSeriesOperations implements Taylor series arithmetic,
where a Taylor series is represented by a stream of its coefficients.

Problems raising a UTS to a negative integer power.

The code in powern(rn,x) which raises an unnecessary error
where no distinction between rational and integer powers are made.

The fix is easy. Since the problem does not exist in SUPS we can
just take the definition there.

See Also:
c )show StreamTaylorSeriesOperations
StreamTaylorSeriesOperations (STTAYLOR)

Exports:
- addiag
- coerce
- compose
- deriv
- eval
- evenlambert
- exquo
- gderiv
- generalLambert
- int
- integrate
- invmulti
- lagrange
- lambert
- lazyGintegrate
- lazyIntegrate
- mapdiv
- mapmult
- monom
- multisect
- nide
- oddIntegers
- oddlambert
- power
- multisect
- powern
- recip
- revert
- ?**?
- ?+?
- ?-?
- ?/?

package STTAYLOR StreamTaylorSeriesOperations

StreamTaylorSeriesOperations(A): Exports == Implementation where
A : Ring
RN ==> Fraction Integer
I ==> Integer
NNI ==> NonNegativeInteger
ST ==> Stream
SP2 ==> StreamFunctions2
SP3 ==> StreamFunctions3
L ==> List
LA ==> List A
YS ==> Y$ParadoxicalCombinatorsForStreams(A)
UN ==> Union(ST A,"failed")
Exports ==> with
   "+": (ST A,ST A) -> ST A
   ++ a + b returns the power series sum of \spad{a} and \spad{b}:
   ++ \spad{[a0,a1,...] + [b0,b1,...] = [a0 + b0,a1 + b1,...]}
"-": (ST A,ST A) -> ST A
   ++ a - b returns the power series difference of \spad{a} and \spad{b}:
   ++ \spad{[a0,a1,...] - [b0,b1,...] = [a0 - b0,a1 - b1,...]}
"-": ST A -> ST A
   ++ - a returns the power series negative of \spad{a}:
   ++ \spad{[a0,a1,...] = [- a0,- a1,...]}
"*": (ST A,ST A) -> ST A
   ++ a * b returns the power series (Cauchy) product of \spad{a} and \spad{b}:
   ++ \spad{[a0,a1,...] * [b0,b1,...] = [c0,c1,...]} where
   ++ \spad{ck = sum(i + j = k,ai * bk)}.
"*": (A,ST A) -> ST A
   ++ r * a returns the power series scalar multiplication of \spad{r} by \spad{a}:
   ++ \spad{r * [a0,a1,...] = [r * a0,r * a1,...]}
"*": (ST A,A) -> ST A
   ++ a * r returns the power series scalar multiplication of \spad{a} by \spad{r}:
   ++ \spad{[a0,a1,...] * r = [a0 * r,a1 * r,...]}
"exquo": (ST A,ST A) -> Union(ST A,"failed")
   ++ exquo(a,b) returns the power series quotient of \spad{a} by \spad{b},
   ++ if the quotient exists, and "failed" otherwise
"/": (ST A,ST A) -> ST A
   ++ a / b returns the power series quotient of \spad{a} by \spad{b}.
   ++ An error message is returned if \spad{b} is not invertible.
   ++ This function is used in fixed point computations.
recip : ST A -> UN
   ++ recip(a) returns the power series reciprocal of \spad{a}, or
   ++ "failed" if not possible.
monom : (A,I) -> ST A
   ++ monom(deg,coef) is a monomial of degree deg with coefficient
   ++ coef.
integers : I -> ST I
   ++ integers(n) returns \spad{[n,n+1,n+2,...]}
oddintegers : I -> ST I
   ++ oddintegers(n) returns \spad{[n,n+2,n+4,...]}
int : A -> ST A
   ++ int(r) returns \spad{[r,r+1,r+2,...]}, where \spad{r} is a ring element.
mapmult : (ST A,ST A) -> ST A
   ++ mapmult([a0,a1,...],[b0,b1,...]) returns \spad{[a0*b0,a1*b1,...]}
deriv : ST A -> ST A
   ++ deriv(a) returns the derivative of the power series with
   ++ respect to the power series variable. Thus
   ++ \spad{deriv([a0,a1,a2,...])} returns \spad{[a1,2 a2,3 a3,...]}
gderiv : (I -> A,ST A) -> ST A
   ++ gderiv(f,[a0,a1,a2,...]) returns \spad{[f(0)*a0,f(1)*a1,f(2)*a2,...]}. 
coerce : A -> ST A
++ coerce(r) converts a ring element r to a stream with one element.

eval : (ST A,A) -> ST A
++ eval(a,r) returns a stream of partial sums of the power series
++ \spad{a} evaluated at the power series variable equal to r.

compose : (ST A,ST A) -> ST A
++ compose(a,b) composes the power series \spad{a} with
++ the power series b.

lagrange : ST A -> ST A
++ lagrange(g) produces the power series for f where f is
++ implicitly defined as \spad{f(z) = z*g(f(z))}.

revert : ST A -> ST A
++ revert(a) computes the inverse of a power series \spad{a}
++ with respect to composition.
++ the series should have constant coefficient 0 and first
++ order coefficient 1.

addiag : ST ST A -> ST A
++ addiag(x) performs diagonal addition of a stream of streams. if x =
++ \spad{[\ldots,[a<0,0>,a<0,1>,\ldots],[a<1,0>,a<1,1>,\ldots],[a<2,0>,a<2,1>,\ldots]\ldots]}.
++ and \spad{addiag(x) = \[b<0,b<1>,\ldots\], then b<k> = \sum(i+j=k,a<i,j>)}.

lambert : ST A -> ST A
++ lambert(st) computes \spad{f(x) + f(x**2) + f(x**3) + \ldots}
++ if st is a stream representing \spad{f(x)}.
++ This function is used for computing infinite products.
++ If \spad{f(x)} is a power series with constant coefficient 1 then
++ \spad{prod(f(x**n),n = 1..infinity) = exp(lambert(log(f(x))))}.

oddlambert : ST A -> ST A
++ oddlambert(st) computes \spad{f(x) + f(x**3) + f(x**5) + \ldots}
++ if st is a stream representing \spad{f(x)}.
++ This function is used for computing infinite products.
++ If f(x) is a power series with constant coefficient 1 then
++ \spad{prod(f(x**(2*n-1)),n=1..infinity) = exp(oddlambert(log(f(x))))}.

evenlambert : ST A -> ST A
++ evenlambert(st) computes \spad{f(x**2) + f(x**4) + f(x**6) + \ldots}
++ if st is a stream representing \spad{f(x)}.
++ This function is used for computing infinite products.
++ If \spad{f(x)} is a power series with constant coefficient 1, then
++ \spad{prod(f(x**(2*n)),n=1..infinity) = exp(evenlambert(log(f(x))))}.

generalLambert : (ST A,I,I) -> ST A
++ generalLambert(f(x),a,d) returns
++ \spad{f(x**a) + f(x**(a + d)) + f(x**(a + 2 d)) + \ldots}.
++ \spad{f(x)} should have zero constant
++ coefficient and \spad{a} and \spad{d} should be positive.

multisect : (I,I,ST A) -> ST A
++ multisect(a,b,st)
++ selects the coefficients of \spad{x**((a+b)*n+a)},
++ and changes them to \spad{x**n}.

invmultisect : (I,I,ST A) -> ST A
++ invmultisect(a,b,st) substitutes \spad{x**((a+b)*n)} for \spad{x**n}
++ and multiplies by \spad{x**b}.
if A has Algebra RN then
  integrate : (A,ST A) -> ST A
  ++ integrate(r,a) returns the integral of the power series \spad{a}
  ++ with respect to the power series variable integration where
  ++ r denotes the constant of integration. Thus
  ++ \spad{\text{integrate}(a,[a0,a1,a2,\ldots]) = [a,a0,a1/2,a2/3,\ldots]}.
lazyIntegrate : (A,() -> ST A) -> ST A
  ++ lazyIntegrate(r,f) is a local function
  ++ used for fixed point computations.
nlde : ST ST A -> ST A
  ++ nlde(u) solves a
  ++ first order non-linear differential equation described by u of the
  ++ form \spad{[[b<0,0>,b<0,1>,\ldots],[b<1,0>,b<1,1>,\ldots],\ldots]}.
  ++ the differential equation has the form
  ++ \spad{y'=\text{sum}(i=0 \text{ to infinity},j=0 \text{ to infinity},b<i,j>*(x**i)*(y**j))}.
powern : (RN,ST A) -> ST A
  ++ powern(r,f) raises power series f to the power r.
if A has Field then
  mapdiv : (ST A,ST A) -> ST A
  ++ mapdiv([a0,a1,\ldots],[b0,b1,\ldots]) returns
  ++ \spad{[a0/b0,a1/b1,\ldots]}.
lazyGintegrate : (I -> A,A,() -> ST A) -> ST A
  ++ lazyGintegrate(f,r,g) is used for fixed point computations.
power : (A,ST A) -> ST A
  ++ power(a,f) returns the power series f raised to the power \spad{a}.

Implementation ==> add

--% definitions

zro: () -> ST A
  -- returns a zero power series
  zro() == empty()$ST(A)

--% arithmetic

x + y == delay
  empty? y => x
  empty? x => y
  eq?(x,rst x) => map(z +-> frst x+z, y)
  eq?(y,rst y) => map(z +-> z-frst y, x)
  concat(frst x + frst y,rst x + rst y)

x - y == delay
  empty? y => x
  empty? x => -y
  eq?(x,rst x) => map(z +-> frst x-z, y)
  eq?(y,rst y) => map(z +-> z-frst y, x)
  concat(frst x - frst y,rst x - rst y)
\[-y = \text{map}(z \mapsto -z, y)\]

\[(x: ST A) \ast (y: ST A) = \text{delay}\]
\[
\begin{align*}
\text{empty? } y & \Rightarrow \text{zro()} \\
\text{empty? } x & \Rightarrow \text{zro()} \\
\text{concat}(\text{frst } x \ast \text{frst } y, \text{frst } x \ast \text{rst } y + \text{rst } x \ast y)
\end{align*}
\]

\[(s: A) \ast (x: ST A) = \]
\[
\begin{align*}
\text{zero? } s & \Rightarrow \text{zro()} \\
\text{map}(z \mapsto s*z, x)
\end{align*}
\]

\[(x: ST A) \ast (s: A) = \]
\[
\begin{align*}
\text{zero? } s & \Rightarrow \text{zro()} \\
\text{map}(z \mapsto z*s, x)
\end{align*}
\]

\[\text{iDiv: } (ST A, ST A, A) \rightarrow ST A\]
\[\text{iDiv}(x, y, ry0) = \text{delay}\]
\[
\begin{align*}
\text{empty? } x & \Rightarrow \text{empty()} \\
c0 & := \text{frst } x \ast ry0 \\
\text{concat}(c0, \text{iDiv}(\text{rst } x - c0 \ast \text{rst } y, y, ry0))
\end{align*}
\]

\[x \text{ exquo } y = \]
\[
\text{for } n \text{ in } 1.. \text{repeat} \\
\begin{align*}
n > 1000 & \Rightarrow \text{return } "\text{failed}" \\
\text{empty? } y & \Rightarrow \text{return } "\text{failed}" \\
\text{empty? } x & \Rightarrow \text{return empty()} \\
\text{frst } y = 0 & \Rightarrow \\
\text{frst } x = 0 & \Rightarrow (x := \text{rst } x; y := \text{rst } y) \\
\text{return } "\text{failed}" \\
\text{leave } "\text{first entry in y is non-zero}"
\end{align*}
\]
\[
\begin{align*}
(ry0 := \text{recip frst } y) & \text{ case } "\text{failed}" \Rightarrow "\text{failed}" \\
\text{empty? } \text{rst } y & \Rightarrow \text{map}(z \mapsto z*(ry0 :: A), x) \\
\text{iDiv}(x, y, ry0 :: A)
\end{align*}
\]

\[(x: ST A) / (y: ST A) = \text{delay}\]
\[
\begin{align*}
\text{empty? } y & \Rightarrow \text{error } "\div: \text{division by zero}" \\
\text{empty? } x & \Rightarrow \text{empty()} \\
(ry0 := \text{recip frst } y) & \text{ case } "\text{failed}" \Rightarrow \\
\text{error } "\div: \text{second argument is not invertible}" \\
\text{empty? } \text{rst } y & \Rightarrow \text{map}(z \mapsto z*(ry0::A), x) \\
\text{iDiv}(x, y, ry0 :: A)
\end{align*}
\]

\[\text{recip } x = \]
\[
\begin{align*}
\text{empty? } x & \Rightarrow "\text{failed}"
\text{rhi} & := \text{recip frst } x \\
\text{rhi} & \text{ case } "\text{failed}" \Rightarrow "\text{failed}"
\text{rh} & := \text{rhi :: A} \\
\text{delay} \\
\text{concat(rh, \text{iDiv}(\text{rh} \ast \text{rst } x, x, \text{rh}))}
\end{align*}
\]
% coefficients

rp: (I,A) -> L A
-- rp(z,s) is a list of length z each of whose entries is s.
rp(z,s) ==
  z <= 0 => empty()
  concat(s,rp(z-1,s))

rpSt: (I,A) -> ST A
-- rpSt(z,s) is a stream of length z each of whose entries is s.
rpSt(z,s) == delay
  z <= 0 => empty()
  concat(s,rpSt(z-1,s))

monom(s,z) ==
  z < 0 => error "monom: cannot create monomial of negative degree"
  concat(rpSt(z,0),concat(s,zro()))

% some streams of integers
nnintegers: NNI -> ST NNI
nnintegers zz == generate(y +-> y+1, zz)
integers z == generate(y +-> y+1, z)
oddintegers z == generate(y +-> y+2, z)
int s == generate(y +-> y+1, s)

% derivatives

mapmult(x,y) == delay
  empty? y => zro()
  empty? x => zro()
  concat(frst x * frst y,mapmult(rst x,rst y))

deriv x ==
  empty? x => zro()
  mapmult(int 1,rest x)

gderiv(f,x) ==
  empty? x => zro()
  mapmult(map(f,integers 0)$SP2(I,A),x)

% coercions

coerce(s:A) ==
  zero? s => zro()
  concat(s,zro())

% evaluations and compositions

eval(x,at) ==
  scan(0,(y,z) +-> y+z,mapmult(x,generate(y +-> at*y,1)))$SP2(A,A)
compose(x,y) == delay
    empty? y => concat(frst x,zro())
    not zero? frst y =>
        error "compose: 2nd argument should have 0 constant coefficient"
    empty? x => zro()
    concat(frst x,compose(rst x,y) * rst(y))

--% reversion
lagrangere:(ST A,ST A) -> ST A
lagrangere(x,c) == delay(concat(0,compose(x,c)))
lagrange x == YS(y +-> lagrangere(x,y))
revert x ==
    empty? x => error "revert should start 0,1,..."
    zero? frst x =>
        empty? rst x => error "revert: should start 0,1,..."
        one? frst rst x => lagrange(recip(rst x) :: (ST A))
    (frst rst x) = 1 => lagrange(recip(rst x) :: (ST A))
    error "revert: should start 0,1,..."

--% lambert functions
addiag(ststa:ST ST A) == delay
    empty? ststa => zro()
    empty? frst ststa => concat(0,addiag rst ststa)
    concat(frst(frst ststa),rst(frst ststa) + addiag(rst ststa))

-- lambert operates on a series +/[a[i]*x**i for i in 1..] , and produces
-- the series +/[a[i]*(x**i/(1-x**i)) for i in 1..] i.e. forms the
-- coefficients A[n] which is the sum of a[i] for all divisors i of n
-- (including 1 and n)

rptg1:(I,A) -> ST A
-- ---------
-- returns the repeating stream [s,0,...,0]; (there are z zeroes)
rptg1(z,s) == repeating concat(s,rp(z,0))
rptg2:(I,A) -> ST A
-- ---------
-- returns the repeating stream [0,...,0,s,0,...,0]
-- there are z leading zeroes and z-1 in the period
rptg2(z,s) == repeating concat(rp(z,0),concat(s,rp(z-1,0)))
rptg3:(I,I,I,A) -> ST A
rptg3(a,d,n,s) ==
    concat(rpSt(n*(a-1),0),repeating(concat(s,rp(d*n-1,0))))
lambert x == delay
empty? x => zro()
zero? frst x =>
    concat(0,addiag(map(rptg1,integers 0,rst x)$SP3(I,A,ST A)))
error "lambert: constant coefficient should be zero"

oddlambert x == delay
empty? x => zro()
zero? frst x =>
    concat(0,addiag(map(rptg1,oddintegers 1,rst x)$SP3(I,A,ST A)))
error "oddlambert: constant coefficient should be zero"

evenlambert x == delay
empty? x => zro()
zero? frst x =>
    concat(0,addiag(map(rptg2,integers 1,rst x)$SP3(I,A,ST A)))
error "evenlambert: constant coefficient should be zero"

generallambert(st,a,d) == delay
a < 1 or d < 1 =>
    error "generallambert: both integer arguments must be positive"
empty? st => zro()
zero? frst st =>
    concat(0,addiag(map((x,y) +-> rptg3(a,d,x,y),
        integers 1,rst st)$SP3(I,A,ST A)))
error "generallambert: constant coefficient should be zero" 

-- misc. functions

ms: (I,I,ST A) -> ST A
ms(m,n,s) == delay
empty? s => zro()
zero? n => concat(frst s,ms(m,m-1,rst s))
ms(m,n-1,rst s)

multisect(b,a,x) == ms(a+b,0,rest(x,a :: NNI))

altn: (ST A,ST A) -> ST A
altn(zs,s) == delay
empty? s => zro()
concat(frst s,concat(zs,altn(zs,rst s)))

invmultisect(a,b,x) ==
    concat(rpSt(b,0),altn(rpSt(a + b - 1,0),x))

-- comps(ststa,y) forms the composition of +/b[i,j]*y**i*x**j
-- where y is a power series in y.

cssa ==> concat$(ST ST A)
mapsa ==> map$SP2(ST A,ST A)
comps: (ST ST A,ST A) -> ST ST A
comps(ststa,x) == delay$(ST ST A)
empty? ststa => empty$((ST ST A)
empty? x => cssa(frst ststa,empty$((ST ST A))
cssa(frst ststa,mapsa(y +-> (rst x)*y,comps(rst ststa,x)))

if A has Algebra RN then
  integre: (ST A,I) -> ST A
  integre(x,n) == delay
    empty? x => zro()
    concat((1$I/n) * frst(x),integre(rst x,n + 1))

integ: ST A -> ST A
integ x == integre(x,1)

integrate(a,x) == concat(a,integ x)
lazyIntegrate(s,xf) == concat(s,integ(delay xf))

nlde: (ST ST A,ST A) -> ST A
nlde(lslsa,c) == lazyIntegrate(0,addiag(comps(lslsa,c)))
nlde lslsa == YS(y +-> nlde(lslsa,y))

RATPOWERS : Boolean := A has "**": (A,RN) -> A

smult: (RN,ST A) -> ST A
smult(rn,x) == map(y +-> rn*y, x)
powerrn: (RN,ST A,ST A) -> ST A
powerrn(rn,x,c) == delay
  concat(1,integ(smult(rn + 1,c * deriv x)) - rst x * c)
powern(rn,x) ==
  order : I := 0
  for n in 0.. repeat
    empty? x => return zro()
    not zero? frst x => (order := n; leave x)
    x := rst x
    n = 1000 =>
      error "**: series with many leading zero coefficients"
    (ord := (order exquo denom(rn))) case "failed" =>
      error "**: rational power does not exist"
    co := frst x
    (invCo := recip co) case "failed" =>
      error "** rational power of coefficient undefined"
    -- This error message is misleading, isn’t it? see sups.spad/cRationalPower
    power :=
      one? co => YS(y +-> powerrn(rn,x,y))
      (co = 1) => YS(y +-> powerrn(rn,x,y))
      (denom rn) = 1 =>
        not negative?(num := numer rn) =>
        -- It seems that this cannot happen, but I don’t know why
        (co**num::NNI) * YS(y +-> powerrn(rn,(invCo :: A) * x, y))
        (invCo :: A)**((-num)::NNI) * YS(y +-> powerrn(rn,(invCo :: A)*x, y))
RATPOWERS => co**rn * YS(y +-> powerrn(rn,(invCo :: A)*x, y))
error "** rational power of coefficient undefined"

if A has Field then
mapdiv(x,y) == delay
empty? y => error "stream division by zero"
empty? x => zro()
concat(frst x/frst y,mapdiv(rst x,rst y))

 ginteg: (I -> A,ST A) -> ST A
ginteg(f,x) == mapdiv(x,map(f,integers 1)$SP2(I,A))

lazyGintegrate(fntoa,s,xf) == concat(s,ginteg(fntoa,delay xf))

finteg: ST A -> ST A
finteg x == mapdiv(x,int 1)
powerre: (A,ST A,ST A) -> ST A
powerre(s,x,c) == delay
empty? x => zro()
frst x=1 => error "**:constant coefficient should be 1"
concat(frst x,finteg((s+1)*(c*deriv x))-rst x * c)
power(s,x) == YS(y +-> powerre(s,x,y))

——

STTAYLOR.dotabb

"STTAYLOR" [color="#F48888" ,href="bookvol10.4.pdf#nameddest=STTAYLOR"]
"FIELD" [color="#4488FF" ,href="bookvol10.2.pdf#nameddest=FIELD"]
"STTAYLOR" -> "FIELD"

——

package STNSR StreamTensor

— StreamTensor.input —

)set break resume
)sys rm -f StreamTensor.output
)spool StreamTensor.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show StreamTensor
--R
--R StreamTensor(R: Type) is a package constructor
--R Abbreviation for StreamTensor is STNSR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STNSR
--R
--R------------------------------- Operations --------------------------------
--R tensorMap : (Stream(R),(R -> List(R))) -> Stream(R)
--R
--E 1
)
)
)spool
)lisp (bye)

—— StreamTensor.help ——

====================================================================
StreamTensor examples
====================================================================

This package has no description

See Also:
o )show StreamTensor

——

StreamTensor (STNSR)
Exports:
tensorMap

— package STNSR StreamTensor —

)abbrev package STNSR StreamTensor
++ Description:
++ This package has no description
StreamTensor(R: Type): with

    tensorMap: (Stream R, R -> List R) -> Stream R
    ++ tensorMap([s1, s2, ...], f) returns the stream consisting of all
    ++ elements of f(s1) followed by all elements of f(s2) and so on.

    == add

tensorMap(s, f) ==
empty? s => empty()
concat([f first s], delay tensorMap(rest s, f))

— STNSR.dotabb —

"STNSR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STNSR"]
"TYPE" [color=lightblue,href="bookvol10.2.pdf#nameddest=TYPE"];
"STNSR" -> "TYPE"

— STTF StreamTranscendentalFunctions —

package STTF StreamTranscendentalFunctions

— StreamTranscendentalFunctions.input —

)set break resume
)sys rm -f StreamTranscendentalFunctions.output
)spool StreamTranscendentalFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamTranscendentalFunctions
StreamTranscendentalFunctions (STTF)

Exports:
  acos  acot  acsc  asec  asin
  atan  cos  cot  csc  exp
  sec  sin  tan  sincos  sinhcos

---

package STTF StreamTranscendentalFunctions ---

)abbrev package STTF StreamTranscendentalFunctions
StreamTranscendentalFunctions(Coef): Exports == Implementation where
Coef : Algebra Fraction Integer
L ==> List
I ==> Integer
RN ==> Fraction Integer
SG ==> String
ST ==> Stream Coef
STT ==> StreamTaylorSeriesOperations Coef
YS ==> ¥$ParadoxicalCombinatorsForStreams(Coef)
Exports == with
-% Exponentials and Logarithms
exp : ST -> ST
  ++ exp(st) computes the exponential of a power series st.
log : ST -> ST
  ++ log(st) computes the log of a power series.
"**" : (ST,ST) -> ST
  ++ st1 ** st2 computes the power of a power series st1 by another
  ++ power series st2.

-% TrigonometricFunctionCategory
sincos : ST -> Record(sin:ST, cos:ST)
  ++ sincos(st) returns a record containing the sine and cosine
  ++ of a power series st.
sin : ST -> ST
  ++ sin(st) computes sine of a power series st.
cos : ST -> ST
  ++ cos(st) computes cosine of a power series st.
tan : ST -> ST
  ++ tan(st) computes tangent of a power series st.
cot : ST -> ST
  ++ cot(st) computes cotangent of a power series st.
sec : ST -> ST
  ++ sec(st) computes secant of a power series st.
csc : ST -> ST
  ++ csc(st) computes cosecant of a power series st.
asin : ST -> ST
  ++ asin(st) computes arcsine of a power series st.
acos : ST -> ST
  ++ acos(st) computes arccosine of a power series st.
atan : ST -> ST
  ++ atan(st) computes arctangent of a power series st.
acot : ST -> ST
++ acot(st) computes arccotangent of a power series st.
asec : ST -> ST
++ asec(st) computes arcsecant of a power series st.
acsc : ST -> ST
++ acsc(st) computes arccosecant of a power series st.

--% HyperbolicTrigonometricFunctionCategory
sinhcosh : ST -> Record(sinh:ST, cosh:ST)
++ sinhcosh(st) returns a record containing
++ the hyperbolic sine and cosine
++ of a power series st.
sinh : ST -> ST
++ sinh(st) computes the hyperbolic sine of a power series st.
cosh : ST -> ST
++ cosh(st) computes the hyperbolic cosine of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
tanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ tanhcosh(st) returns a record containing
++ the hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanh : ST -> ST
++ teiltanh(st) computes the inverse hyperbolic tangent
++ of a power series st.
tanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ tanhcosh(st) returns a record containing
++ the hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanh : ST -> ST
++ teiltanh(st) computes the inverse hyperbolic tangent
++ of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
teiltanhcosh : ST -> Record(tanh:ST, cosh:ST)
++ teiltanhcosh(st) returns a record containing
++ the inverse hyperbolic tangent and cosine
++ of a power series st.
CHAPTER 20. CHAPTER S

NPOWERS : SG := "series expansion has terms of negative degree"
FPowers : SG := "series expansion has terms of fractional degree"
MAYFPOW : SG := "series expansion may have terms of fractional degree"
LOGS : SG := "series expansion has logarithmic term"
NPOWLOG : SG :=
   "series expansion has terms of negative degree or logarithmic term"
FPWLOG : SG :=
   "series expansion has terms of fractional degree or logarithmic term"
NOTINV : SG := "leading coefficient not invertible"

--% Exponentials and Logarithms

expre:(Coef,ST,ST) -> ST
expre(r,e,dx) == lazyIntegrate(r,e*dx)

exp z ==
empty? z => 1 :: ST
(coef := frst z) = 0 => YS(y +-> expre(1,y,deriv z))
TRANSFCN => YS(y +-> expre(exp coef,y,deriv z))
error concat("exp: ",TRCONST)

log z ==
empty? z => error "log: constant coefficient should not be 0"
(coef := frst z) = 0 => error "log: constant coefficient should not be 0"
coef = 1 => lazyIntegrate(0,deriv z/z)
TRANSFCN => lazyIntegrate(log coef,deriv z/z)
error concat("log: ",TRCONST)

z1:ST ** z2:ST == exp(z2 * log z1)

--% Trigonometric Functions

sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
sincosre(rs,rc,sc,dx,sign) ==
[lazyIntegrate(rs,(second sc)*dx),lazyIntegrate(rc,sign*(first sc)*dx)]

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:
--
-- sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
-- sincosre(rs,rc,sc,dx,sign) ==
--   -- st1 : ST := (second sc) * dx
--   -- st2 : ST := (first sc) * dx
--   -- st2 := sign * st2
--   -- [lazyIntegrate(rs,st1),lazyIntegrate(rc,sign*st2)]

sincos z ==
empty? z => [0 :: ST,1 :: ST]
l :=
(coef := frst z) = 0 => YS(y +-> sincosre(0,1,y,deriv z,-1),2)
TRANSFCN => YS(y +-> sincosre(sin coef, cos coef, y, deriv z, -1), 2)
   error concat("sincos: ", TRCONST)
[first 1, second 1]

sin z == sincos(z).sin
cos z == sincos(z).cos

tanre:(Coef, ST, ST, Coef) -> ST
tanre(r, t, dx, sign) == lazyIntegrate(r, ((1 :: ST) + sign * t * t) * dx)

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:

-- tanre:(Coef, ST, ST, Coef) -> ST
-- tanre(r, t, dx, sign) ==
--   st1 : ST := t * t
--   st1 := sign * st1
--   st2 : ST := 1 :: ST
--   st1 := st2 + st1
--   st1 := st1 * dx
--   lazyIntegrate(r, st1)

tan z ==
   empty? z => 0 :: ST
   (coef := frst z) = 0 => YS(y +-> tanre(0, y, deriv z, 1))
   TRANSFCN => YS(y +-> tanre(tan coef, y, deriv z, 1))
   error concat("tan: ", TRCONST)

cotre:(Coef, ST, ST) -> ST
cotre(r, t, dx) == lazyIntegrate(r, -((1 :: ST) + t * t) * dx)

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:

-- cotre:(Coef, ST, ST) -> ST
-- cotre(r, t, dx) ==
--   st1 : ST := t * t
--   st2 : ST := 1 :: ST
--   st1 := st2 + st1
--   st1 := st1 * dx
--   st1 := -st1
--   lazyIntegrate(r, st1)
cot z ==
   empty? z => error "cot: cot(0) is undefined"
   (coef := frst z) = 0 => error concat("cot: ", NPOWERS)
   TRANSFCN => YS(y +-> cotre(cot coef, y, deriv z))
   error concat("cot: ", TRCONST)

sec z ==
empty? z => 1 :: ST
first z = 0 => recip(cos z) :: ST
TRANSFCN =>
  cosz := cos z
  first cosz = 0 => error concat("sec: ",NPOWERS)
  recip(cosz) :: ST
error concat("sec: ",TRCONST)

csc z ==
empty? z => error "csc: csc(0) is undefined"
TRANSFCN =>
  sinz := sin z
  first sinz = 0 => error concat("csc: ",NPOWERS)
  recip(sinz) :: ST
error concat("csc: ",TRCONST)

orderOrFailed : ST -> Union(I,"failed")
orderOrFailed x ==
  -- returns the order of x or "failed"
  -- if -1 is returned, the series is identically zero
  for n in 0..1000 repeat
    empty? x => return -1
    not zero? frst x => return n :: I
    x := rst x
  "failed"

asin z ==
empty? z => 0 :: ST
(coef := frst z) = 0 =>
  integrate(0,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
TRANSFCN =>
  coef = 1 or coef = -1 =>
    x := (1 :: ST) - z*z
    -- compute order of 'x'
    (ord := orderOrFailed x) case "failed" =>
      error concat("asin: ",MAYFPow)
    (order := ord :: I) = -1 => return asin(coef) :: ST
    odd? order => error concat("asin: ",FPOWERS)
    squirt := powern(1/2,x)
    (quot := (deriv z) exquo squirt) case "failed" =>
      error concat("asin: ",NUTINV)
    integrate(asin coef,quot :: ST)
    integrate(asin coef,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
  error concat("asin: ",TRCONST)

acos z ==
empty? z =>
  TRANSFCN => acos(0)$Coef :: ST
  error concat("acos: ",TRCONST)
TRANSFCN =>
coef := frst z
coef = 1 or coef = -1 =>
x := (1 :: ST) - z*z
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("acos: ",MAYFPOW)
(order := ord :: I) = -1 => return acos(coef) :: ST
odd? order => error concat("acos: ",FPowers)
squirt := powern(1/2,x)
(quot := (-deriv z) exquo squirt) case "failed" =>
  error concat("acos: ",NOTINV)
integrate(acos coef,quot :: ST)
integrate(acos coef,-powern(-1/2,(1 :: ST) - z*z) * (deriv z))
error concat("acos: ",TRCONST)

atan z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 =>
    integrate(0, (recip((1 :: ST) + z*z) :: ST) * (deriv z))
TRANSFCN =>
  (y := recip((1 :: ST) + z*z)) case "failed" =>
    error concat("atan: ",LOGS)
  integrate(atan coef,(y :: ST) * (deriv z))
error concat("atan: ",TRCONST)

acot z ==
  empty? z =>
    TRANSFCN => acot(0)$Coef :: ST
    error concat("acot: ",TRCONST)
TRANSFCN =>
  (y := recip((1 :: ST) + z*z)) case "failed" =>
    error concat("acot: ",LOGS)
  integrate(acot frst z,-(y :: ST) * (deriv z))
error concat("acot: ",TRCONST)

asec z ==
  empty? z => error "asec: constant coefficient should not be 0"
TRANSFCN =>
  (coef := frst z) = 0 =>
    error "asec: constant coefficient should not be 0"
  coef = 1 or coef = -1 =>
x := z*z - (1 :: ST)
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("asec: ",MAYFPOW)
(order := ord :: I) = -1 => return asec(coef) :: ST
odd? order => error concat("asec: ",FPowers)
squirt := powern(1/2,x)
(quot := (deriv z) exquo squirt) case "failed" =>
  error concat("asec: ",NOTINV)
\[
\text{acsc } z =
\]
\begin{itemize}
\item empty? \( z \) \Rightarrow \text{error "acsc: constant coefficient should not be zero"}
\item TRANSFCN \Rightarrow
\begin{itemize}
\item \((\text{coef} := \text{frst } z) = 0 \Rightarrow \text{error "acsc: constant coefficient should not be zero"}\)
\item coef = 1 or coef = -1 \Rightarrow
\begin{itemize}
\item \( x := z^2 - (1 :: \text{ST}) \)
\item -- compute order of 'x'
\item \((\text{ord} := \text{orderOrFailed } x) \) case "failed" \Rightarrow
\begin{itemize}
\item \text{error concat("acsc: ",MAYFPOW)}
\item \text{error concat("acsc: ",NOTINV)}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\]\(\text{error concat("acsc: ",TRCONST)}\)

\text{acsc } z ==
\begin{itemize}
\item empty? \( z \) \Rightarrow \text{error "acsc: constant coefficient should not be zero"}
\item TRANSFCN \Rightarrow
\begin{itemize}
\item \((\text{coef} := \text{frst } z) = 0 \Rightarrow \text{error "acsc: constant coefficient should not be zero"}\)
\item coef = 1 or coef = -1 \Rightarrow
\begin{itemize}
\item \( x := z^2 - (1 :: \text{ST}) \)
\item -- compute order of 'x'
\item \((\text{ord} := \text{orderOrFailed } x) \) case "failed" \Rightarrow
\begin{itemize}
\item \text{error concat("acsc: ",MAYFPOW)}
\item \text{error concat("acsc: ",NOTINV)}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\]\(\text{error concat("acsc: ",TRCONST)}\)

\text{--% Hyperbolic Trigonometric Functions}

\text{sinhcosh } z ==
\begin{itemize}
\item empty? \( z \) \Rightarrow [0 :: \text{ST},1 :: \text{ST}]
\item TRANSFCN \Rightarrow \text{YS} \( y \mapsto \text{sincosre}(0,1,y,\text{deriv } z,1),2 \)
\item \text{error concat("sinhcosh: ",TRCONST)}
\end{itemize}
\]\[\text{[first l,second l]}\]

\text{sinh } z == \text{sinhcosh}(z).\text{sinh}
\text{cosh } z == \text{sinhcosh}(z).\text{cosh}

\text{tanh } z ==
\begin{itemize}
\item empty? \( z \) \Rightarrow 0 :: \text{ST}
\item TRANSFCN \Rightarrow \text{YS} \( y \mapsto \text{tanre}(0,y,\text{deriv } z,-1))\)
\item \text{error concat("tanh: ",TRCONST)}
\end{itemize}

\text{coth } z ==
\begin{itemize}
\item \text{tanhz} := \text{tanh } z
\item empty? \text{tanhz} \Rightarrow \text{error "coth: coth(0) is undefined"}
\item \( \text{frst } \text{tanhz} = 0 \Rightarrow \text{error concat("coth: ",NPowers)}\)
\end{itemize}
recip(tanhz) :: ST

sech z ==
  coshz := cosh z
  (empty? coshz) or (frst coshz = 0) => error concat("sech: ",NPowers)
  recip(coshz) :: ST

csch z ==
  sinhz := sinh z
  (empty? sinhz) or (frst sinhz = 0) => error concat("csch: ",NPowers)
  recip(sinhz) :: ST

asinh z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 => log(z + powern(1/2,(1 :: ST) + z*z))
  TRANSFCN =>
    x := (1 :: ST) + z*z
    -- compute order of 'x', in case coefficient(z,0) = +- %i
    (ord := orderOrFailed x) case "failed" =>
      error concat("asinh: ",MAYFPow)
    (order := ord :: I) = -1 => return asinh(coef) :: ST
    odd? order => error concat("asinh: ",FPowers)
    -- the argument to 'log' must have a non-zero constant term
    log(z + powern(1/2,x))
  error concat("asinh: ",TRConst)

acosh z ==
  empty? z =>
    TRANSFCN => acosh(0)$Coef :: ST
    error concat("acosh: ",TRConst)
  TRANSFCN =>
    coef := frst z
    coef = 1 or coef = -1 =>
      x := z*z - (1 :: ST)
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("acosh: ",MAYFPow)
      (order := ord :: I) = -1 => return acosh(coef) :: ST
      odd? order => error concat("acosh: ",FPowers)
      -- the argument to 'log' must have a non-zero constant term
      log(z + powern(1/2,x))
    log(z + powern(1/2,z*z - (1 :: ST)))
  error concat("acosh: ",TRConst)

atanh z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 =>
    (inv(2::RN)::Coef) * log(((1 :: ST) + z)/((1 :: ST) - z))
  TRANSFCN =>
    coef = 1 or coef = -1 => error concat("atanh: ",LOGS)
(inv(2::RN)::Coef) * log((1 :: ST) + z)/(1 :: ST) - z))
error concat("atanh: ",TRCONST)

acoth z ==
empty? z =>
TRANSFCN => acoth(0)$Coef :: ST
error concat("acoth: ",TRCONST)
TRANSFCN =>
frst z = 1 or frst z = -1 => error concat("acoth: ",LOGS)
(inv(2::RN)::Coef) * log((z + (1 :: ST))/(z - (1 :: ST)))
error concat("acoth: ",TRCONST)

asech z ==
empty? z => error "asech: asech(0) is undefined"
TRANSFCN =>
(coef := frst z) = 0 => error concat("asech: ",NPOWLOG)
coef = 1 or coef = -1 =>
x := (1 :: ST) - z*z
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("asech: ",MAYFPW)
(order := ord :: I) = -1 => return asech(coef) :: ST
odd? order => error concat("asech: ",FPOWERS)
log(((1 :: ST) + powern(1/2,x))/z)
log(((1 :: ST) + powern(1/2,(1 :: ST) - z*z))/z)
error concat("asech: ",TRCONST)

acsch z ==
empty? z => error "acsch: acsch(0) is undefined"
TRANSFCN =>
frst z = 0 => error concat("acsch: ",NPOWLOG)
x := z*z + (1 :: ST)
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("acsch: ",MAYFPW)
(order := ord :: I) = -1 => return acsch(frst z) :: ST
odd? order => error concat("acsch: ",FPOWERS)
log(((1 :: ST) + powern(1/2,x))/z)
error concat("acsch: ",TRCONST)

———

"STTF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STTF"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"STTF" -> "STRING"
package STTFNC StreamTranscendentalFunctionsNonCommutative

StreamTranscendentalFunctionsNonCommutative examples

StreamTranscendentalFunctionsNonCommutative implements transcendental functions on Taylor series over a non-commutative ring, where a Taylor series is represented by a stream of its coefficients.

See Also:
c) show StreamTranscendentalFunctionsNonCommutative
StreamTranscendentalFunctionsNonCommutative (STTFNC)

Exports:
acos acot acsc asec asin
atan cos cot csc exp
sec sin tan ?**?

package STTFNC StreamTranscendentalFunctionsNonCommutative

StreamTranscendentalFunctionsNonCommutative(Coef): _
   Exports == Implementation where
   Coef : Algebra Fraction Integer
   I    ==> Integer
   SG   ==> String
   ST   ==> Stream Coef
   STTF ==> StreamTranscendentalFunctions Coef

Exports ==> with
--% Exponentials and Logarithms
   exp : ST -> ST
   ++ exp(st) computes the exponential of a power series st.
   log : ST -> ST
   ++ log(st) computes the log of a power series.
   "**" : (ST,ST) -> ST
   ++ st1 ** st2 computes the power of a power series st1 by another
   ++ power series st2.
--% TrigonometricFunctionCategory
sin : ST -> ST
++ sin(st) computes sine of a power series st.
cos : ST -> ST
++ cos(st) computes cosine of a power series st.
tan : ST -> ST
++ tan(st) computes tangent of a power series st.
cot : ST -> ST
++ cot(st) computes cotangent of a power series st.
sec : ST -> ST
++ sec(st) computes secant of a power series st.
csc : ST -> ST
++ csc(st) computes cosecant of a power series st.
asin : ST -> ST
++ asin(st) computes arcsine of a power series st.
acos : ST -> ST
++ acos(st) computes arccosine of a power series st.
atan : ST -> ST
++ atan(st) computes arctangent of a power series st.
acot : ST -> ST
++ acot(st) computes arccotangent of a power series st.
asec : ST -> ST
++ asec(st) computes arcsecant of a power series st.
acsc : ST -> ST
++ acsc(st) computes arccosecant of a power series st.

--% HyperbolicTrigonometricFunctionCategory
sinh : ST -> ST
++ sinh(st) computes the hyperbolic sine of a power series st.
cosh : ST -> ST
++ cosh(st) computes the hyperbolic cosine of a power series st.
tanh : ST -> ST
++ tanh(st) computes the hyperbolic tangent of a power series st.
coth : ST -> ST
++ coth(st) computes the hyperbolic cotangent of a power series st.
sech : ST -> ST
++ sech(st) computes the hyperbolic secant of a power series st.
csch : ST -> ST
++ csch(st) computes the hyperbolic cosecant of a power series st.
asinh : ST -> ST
++ asinh(st) computes the inverse hyperbolic sine of a power series st.
acosh : ST -> ST
++ acosh(st) computes the inverse hyperbolic cosine
++ of a power series st.
atanh : ST -> ST
++ atanh(st) computes the inverse hyperbolic tangent
++ of a power series st.
acoth : ST -> ST
++ acoth(st) computes the inverse hyperbolic
cotangent of a power series st.
asech : ST -> ST
  ++ asech(st) computes the inverse hyperbolic secant of a
  ++ power series st.
acsch : ST -> ST
  ++ acsch(st) computes the inverse hyperbolic cosecant of a power series st.

Implementation ==> add
    import StreamTaylorSeriesOperations(Coef)

--% Error Reporting

 ZERO : SG := "series must have constant coefficient zero"
 ONE : SG := "series must have constant coefficient one"
 NPOWERS : SG := "series expansion has terms of negative degree"

--% Exponentials and Logarithms

exp z ==
    empty? z => 1 :: ST
    (frst z) = 0 =>
        expx := exp(monom(1,1))$STTF
        compose(expx,z)
        error concat("exp: ",ZERO)

log z ==
    empty? z => error concat("log: ",ONE)
    (frst z) = 1 =>
        log1PlusX := log(monom(1,0) + monom(1,1))$STTF
        compose(log1PlusX,z - monom(1,0))
        error concat("log: ",ONE)

    (z1:ST) ** (z2:ST) == exp(log(z1) * z2)

--% Trigonometric Functions

sin z ==
    empty? z => 0 :: ST
    (frst z) = 0 =>
        sinx := sin(monom(1,1))$STTF
        compose(sinx,z)
        error concat("sin: ",ZERO)

cos z ==
    empty? z => 1 :: ST
    (frst z) = 0 =>
        cosx := cos(monom(1,1))$STTF
        compose(cosx,z)
        error concat("cos: ",ZERO)
```lisp
PACKAGESTTFNC STREAMTRANSCENDENTALFUNCTIONSNONCOMMUTATIVE3463

tan z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    tanx := tan(monom(1,1))$STTF
    compose(tanx,z)
  error concat("tan: ",ZERO)

cot z ==
  empty? z => error "cot: cot(0) is undefined"
  (frst z) = 0 => error concat("cot: ",NPOWERS)
  error concat("cot: ",ZERO)

sec z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    secx := sec(monom(1,1))$STTF
    compose(secx,z)
  error concat("sec: ",ZERO)

csc z ==
  empty? z => error "csc: csc(0) is undefined"
  (frst z) = 0 => error concat("csc: ",NPOWERS)
  error concat("csc: ",ZERO)

asin z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    asinx := asin(monom(1,1))$STTF
    compose(asinx,z)
  error concat("asin: ",ZERO)

atan z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    atanx := atan(monom(1,1))$STTF
    compose(atanx,z)
  error concat("atan: ",ZERO)

acos z == error "acos: acos undefined on this coefficient domain"
acot z == error "acot: acot undefined on this coefficient domain"
asec z == error "asec: asec undefined on this coefficient domain"
acsc z == error "acsc: acsc undefined on this coefficient domain"

-- % Hyperbolic Trigonometric Functions

sinh z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    sinhx := sinh(monom(1,1))$STTF
    compose(sinhx,z)
```
\begin{verbatim}
error concat("sinh: ",ZERO)

cosh z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    coshx := cosh(monom(1,1))$STTF
    compose(coshx,z)
  error concat("cosh: ",ZERO)

tanh z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    tanhx := tanh(monom(1,1))$STTF
    compose(tanhx,z)
  error concat("tanh: ",ZERO)

coth z ==
  empty? z => error "coth: coth(0) is undefined"
  (frst z) = 0 => error concat("coth: ",NPOWERS)
  error concat("coth: ",ZERO)

sech z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    sechx := sech(monom(1,1))$STTF
    compose(sechx,z)
  error concat("sech: ",ZERO)

csch z ==
  empty? z => error "csch: csch(0) is undefined"
  (frst z) = 0 => error concat("csch: ",NPOWERS)
  error concat("csch: ",ZERO)

asinh z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    asinhx := asinh(monom(1,1))$STTF
    compose(asinhx,z)
  error concat("asinh: ",ZERO)

atanh z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    atanhx := atanh(monom(1,1))$STTF
    compose(atanhx,z)
  error concat("atanh: ",ZERO)

acosh z == error "acosh: acosh undefined on this coefficient domain"
acoth z == error "acoth: acoth undefined on this coefficient domain"
asech z == error "asech: asech undefined on this coefficient domain"
\end{verbatim}
acsch z == error "acsch: acsch undefined on this coefficient domain"

---

— STTFNC.dotabb —

"STTFNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STTFNC"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"STTFNC" -> "STRING"

---

package SCPKG StructuralConstantsPackage

— StructuralConstantsPackage.input —

)set break resume
)sys rm -f StructuralConstantsPackage.output
)spool StructuralConstantsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StructuralConstantsPackage
--E 1

)spool
)lisp (bye)

---

— StructuralConstantsPackage.help —

====================================================================
StructuralConstantsPackage examples
====================================================================

StructuralConstantsPackage provides functions creating structural constants from a multiplication tables or a basis of a matrix algebra and other useful functions in this context.

See Also:
StructuralConstantsPackage (SCPKG)

Exports:
coordinates structuralConstants

--- package SCPKG StructuralConstantsPackage ---

\( \text{StructuralConstantsPackage}(R:\text{Field}) : \text{public} \Rightarrow \text{private} \) where

\[
\begin{align*}
L & \Rightarrow \text{List} \\
S & \Rightarrow \text{Symbol} \\
\text{FRAC} & \Rightarrow \text{Fraction} \\
\text{POLY} & \Rightarrow \text{Polynomial} \\
V & \Rightarrow \text{Vector} \\
M & \Rightarrow \text{Matrix} \\
\text{REC} & \Rightarrow \text{Record(particular: Union(V R,"failed"),basis: List V R)} \\
\text{LSMP} & \Rightarrow \text{LinearSystemMatrixPackage(R,V R,V R, M R)}
\end{align*}
\]

\( \text{public} \Rightarrow \text{with} \)

-- what we really want to have here is a matrix over
-- linear polynomials in the list of symbols, having arbitrary
-- coefficients from a ring extension of $R$, e.g. FRAC POLY $R$.

\texttt{structuralConstants : (L S, M FRAC POLY R) \rightarrow V M FRAC POLY R}
++ \texttt{structuralConstants(ls,mt)} determines the structural constants
++ of an algebra with generators $ls$ and multiplication table $mt$, the
++ entries of which must be given as linear polynomials in the
++ indeterminates given by $ls$. The result is in particular useful
++ as fourth argument for \texttt{spadtype\{AlgebraGivenByStructuralConstants\}}
++ and \texttt{spadtype\{GenericNonAssociativeAlgebra\}}.

\texttt{structuralConstants : (L S, M POLY R) \rightarrow V M POLY R}
++ \texttt{structuralConstants(ls,mt)} determines the structural constants
++ of an algebra with generators $ls$ and multiplication table $mt$, the
++ entries of which must be given as linear polynomials in the
++ indeterminates given by $ls$. The result is in particular useful
++ as fourth argument for \texttt{spadtype\{AlgebraGivenByStructuralConstants\}}
++ and \texttt{spadtype\{GenericNonAssociativeAlgebra\}}.

\texttt{structuralConstants : L M R \rightarrow V M R}
++ \texttt{structuralConstants(basis)} takes the basis of a matrix
++ algebra, e.g. the result of \texttt{spadfun\{basisOfCentroid\}} and calculates
++ the structural constants.
++ Note, that the it is not checked, whether basis really is a
++ basis of a matrix algebra.

\texttt{coordinates : (M R, L M R) \rightarrow V R}
++ \texttt{coordinates(a,[v1,...,vn])} returns the coordinates of \texttt{a}
++ with respect to the \texttt{R}-module basis \texttt{v1},...\texttt{vn}.

\texttt{private ==> add}

\texttt{matrix2Vector : M R \rightarrow V R}

\texttt{matrix2Vector m ==}
  lili : L L R := listofLists m
  li : L R := reduce(concat, lili)
  construct(li)$V R

\texttt{coordinates(x,b) ==}
  m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
  n : NonNegativeInteger := nrows(b.1) * ncols(b.1)
  transitionMatrix : Matrix R := new(n,m,0$R)$Matrix(R)
  for i in 1..m repeat
    setColumn_(!(transitionMatrix,i,matrix2Vector(b.i))
  res : REC := solve(transitionMatrix,matrix2Vector(x))$LSMP
  if (not every?(zero?$R,first res.basis)) then
    error("coordinates: the second argument is linearly dependent")
  (res.particle case "failed") =>
    error("coordinates: first argument is not in linear span of second
    argument")
  (res.particle) :: (Vector R)

\texttt{structuralConstants b ==}
\[ n := \text{rank()} \]

-- be careful with the possibility that \( b \) is not a basis
\[ m : \text{NonNegativeInteger} := (\text{maxIndex } b) :: \text{NonNegativeInteger} \]
\[ \text{sC} : \text{Vector Matrix } R := [\text{new}(m,m,0\$R) \text{ for } k \in 1..m] \]
for \( i \) in 1..m repeat
  for \( j \) in 1..m repeat
    \[ \text{covec} : \text{Vector } R := \text{coordinates}(b.i \ast b.j, b)\% \]
    for \( k \) in 1..m repeat
      \[ \text{setelt}(\text{sC}.k, i, j, \text{covec}.k) \]
\[ \text{sC} \]

\text{structuralConstants}(ls:L S, mt: M POLY R) ==
\[ nn := #(ls) \]
\[ \text{nrows(mt) } ^=` nn \text{ or ncols(mt) } ^=` nn => \]
  \[ \text{error } "\text{structuralConstants: size of second argument does not } \]
  \[ \text{agree with number of generators}" \]
  \[ \text{gamma} : L M POLY R := [] \]
  \[ \text{lscopy} : L S := \text{copy } ls \]
  while not null lscopy repeat
    \[ \text{mat} : M POLY R := \text{new}(nn,nn,0) \]
    \[ s : S := \text{first lscopy} \]
    for \( i \) in 1..nn repeat
      for \( j \) in 1..nn repeat
        \[ p := \text{qelt(mt}, i, j) \]
        totalDegree(p, ls) > 1 =>
          \[ \text{error } "\text{structuralConstants: entries of second argument } \]
          \[ \text{must be linear polynomials in the generators}" \]
        if \( (c := \text{coefficient}(p, s, 1)) ^=` 0 \text{ then } \text{qsetelt}_{!}(\text{mat}, i, j, c) \]
        \[ \text{gamma} := \text{cons(mat, gamma)} \]
      \[ \text{lscopy} := \text{rest lscopy} \]
    \[ \text{vector reverse gamma} \]

\text{structuralConstants}(ls:L S, mt: M FRAC POLY R) ==
\[ nn := #(ls) \]
\[ \text{nrows(mt) } ^=` nn \text{ or ncols(mt) } ^=` nn => \]
  \[ \text{error } "\text{structuralConstants: size of second argument does not } \]
  \[ \text{agree with number of generators}" \]
  \[ \text{gamma} : L M FRAC(POLY R) := [] \]
  \[ \text{lscopy} : L S := \text{copy } ls \]
  while not null lscopy repeat
    \[ \text{mat} : M FRAC(POLY R) := \text{new}(nn,nn,0) \]
    \[ s : S := \text{first lscopy} \]
    for \( i \) in 1..nn repeat
      for \( j \) in 1..nn repeat
        \[ r := \text{qelt(mt}, i, j) \]
        \[ q := \text{denom}(r) \]
        totalDegree(q, ls) ^=` 0 =>
          \[ \text{error } "\text{structuralConstants: entries of second argument } \]
          \[ \text{must be (linear) polynomials in the generators}" \]
        \[ p := \text{numerator}(r) \]
totalDegree(p,ls) > 1 =>
   error "structuralConstants: entries of second argument _
must be linear polynomials in the generators"
   if (c := coefficient(p, s, 1) ) ^= 0 then qsetelt_(mat,i,j,c/q)
   gamma := cons(mat, gamma)
   lscopy := rest lscopy
   vector reverse gamma

——

— SCPKG.dotabb —

"SCPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SCPKG"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SCPKG" -> "PFECAT"

——

package SHP SturmHabichtPackage

— SturmHabichtPackage.input —

)set break resume
)sys rm -f SturmHabichtPackage.output
)spool SturmHabichtPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SturmHabichtPackage
--E 1

)spool
)lisp (bye)

——

— SturmHabichtPackage.help —

====================================================================
SturmHabichtPackage examples
====================================================================
This package produces functions for counting etc. real roots of univariate polynomials in \( x \) over \( \mathbb{R} \), which must be an OrderedIntegralDomain.

See Also:
- \( \text{show SturmHabichtPackage} \)

---

**SturmHabichtPackage (SHP)**

---

**Exports:**
- SturmHabicht
- SturmHabichtCoefficients
- SturmHabichtMultiple
- SturmHabichtSequence
- countRealRoots
- countRealRootsMultiple
- subresultantSequence

---

\( \text{package SHP SturmHabichtPackage} \)
NNI ==> NonNegativeInteger

T ==

-- subresultantSequenceBegin:(UP(x,R),UP(x,R)) -> L UP(x,R)
-- ++ \spad{subresultantSequenceBegin(p1,p2)} computes the initial terms
-- ++ of the Subresultant sequence Sres(j)(P,deg(P),Q,deg(P)-1)
-- ++ when \spad{deg(Q)<deg(P)}
-- subresultantSequenceNext:L UP(x,R) -> L UP(x,R)
-- subresultantSequenceInner:(UP(x,R),UP(x,R)) -> L UP(x,R)
-- subresultantSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
++ subresultantSequence(p1,p2) computes the (standard)
++ subresultant sequence of p1 and p2
-- sign:R -> R
-- delta:NNI -> R
-- polsth1:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
-- polsth2:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
-- polsth3:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
-- SturmHabichtSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
++ SturmHabichtSequence(p1,p2) computes the Sturm-Habicht
++ sequence of p1 and p2
-- SturmHabichtCoefficients:(UP(x,R),UP(x,R)) -> L R
++ SturmHabichtCoefficients(p1,p2) computes the principal
++ Sturm-Habicht coefficients of p1 and p2
-- variation:L R -> INT
-- permanence:L R -> INT
-- qzeros:L R -> L R
-- epsil:(NNI,R,R) -> INT
-- numbnc:L R -> NNI
-- numbce:L R -> NNI
-- wfunctaux:L R -> INT
-- wfunct:L R -> INT

SturmHabicht:(UP(x,R),UP(x,R)) -> INT
++ SturmHabicht(p1,p2) computes c_{+}-c_{-} where
++ c_{+} is the number of real roots of p1 with p2>0 and c_{-}
++ is the number of real roots of p1 with p2<0. If p2=1 what
++ you get is the number of real roots of p1.
countRealRoots:(UP(x,R)) -> INT
++ countRealRoots(p) says how many real roots p has,
if R has GcdDomain then
SturmHabichtMultiple:(UP(x,R),UP(x,R)) -> INT
++ SturmHabichtMultiple(p1,p2) computes c_{+}-c_{-} where
++ c_{+} is the number of real roots of p1 with p2>0 and c_{-}
++ is the number of real roots of p1 with p2<0. If p2=1 what
++ you get is the number of real roots of p1.
countRealRootsMultiple:(UP(x,R)) -> INT
++ countRealRootsMultiple(p) says how many real roots p has,
++ counted with multiplicity
C == add
   p1,p2: UP(x,R)
   Ex ==> OutputForm
   import OutputForm

subresultantSequenceBegin(p1,p2):L UP(x,R) ==
   d1:NNI:=degree(p1)
   d2:NNI:=degree(p2)
   n:NNI:=(d1-1)::NNI
   d2 = n =>
      Pr:UP(x,R):=pseudoRemainder(p1,p2)
      append([p1,p2]::L UP(x,R),[Pr]::L UP(x,R))
   d2 = (n-1)::NNI =>
      Lc1:UP(x,R):=leadingCoefficient(p1)*leadingCoefficient(p2)*p2
      Lc2:UP(x,R):=-leadingCoefficient(p1)*pseudoRemainder(p1,p2)
      append([p1,p2]::L UP(x,R),[Lc1,Lc2]::L UP(x,R))
   LSubr:L UP(x,R):=[p1,p2]
   in1:INT:=(d2+1)::INT
   in2:INT:=(n-1)::INT
   for i in in1..in2 repeat
      LSubr:L UP(x,R):=append(LSubr::L UP(x,R),[0]::L UP(x,R))
      c1:R:=(leadingCoefficient(p1)*leadingCoefficient(p2))**((n-d2)::NNI)
      Lc1:UP(x,R):=monomial(c1,0)*p2
      Lc2:UP(x,R):=
         (-leadingCoefficient(p1))**((n-d2)::NNI)*pseudoRemainder(p1,p2)
      append(LSubr::L UP(x,R),[Lc1,Lc2]::L UP(x,R))

subresultantSequenceNext(LcsI:L UP(x,R)):L UP(x,R) ==
   p2:UP(x,R):=last LcsI
   p1:UP(x,R):=first rest reverse LcsI
   d1:NNI:=degree(p1)
   d2:NNI:=degree(p2)
   in1:NNI:=(d1-1)::NNI
   d2 = in1 =>
      pr1:UP(x,R):=
         (pseudoRemainder(p1,p2) exquo (leadingCoefficient(p1))**2)::UP(x,R)
      append(LcsI:L UP(x,R),[pr1]::L UP(x,R))
   d2 < in1 =>
      c1:R:=leadingCoefficient(p1)
      pr1:UP(x,R):=
         (leadingCoefficient(p2)**((in1-d2)::NNI)*p2 exquo 
           c1**((in1-d2)::NNI))::UP(x,R)
      pr2:UP(x,R):=
         (pseudoRemainder(p1,p2) exquo (-c1)**((in1-d2+2)::NNI))::UP(x,R)
      LSub:L UP(x,R):=[pr1,pr2]
      for k in ((d2+1)::INT)..((in1-1)::INT) repeat
         LSub:L UP(x,R):=append([0]:L UP(x,R),LSub:L UP(x,R))
      append(LcsI:L UP(x,R),LSub:L UP(x,R))
subresultantSequenceInner(p1,p2):L UP(x,R) ==
   Lin:L UP(x,R):=subresultantSequenceBegin(p1:UP(x,R),p2:UP(x,R))
   indf:NNI:= if not(Lin.last::UP(x,R) = 0) then degree(Lin.last::UP(x,R))
               else 0
   while not(indf = 0) repeat
      Lin:L UP(x,R):=subresultantSequenceNext(Lin:L UP(x,R))
      indf:NNI:= if not(Lin.last::UP(x,R)=0) then degree(Lin.last::UP(x,R))
               else 0
      for j in #(Lin:L UP(x,R))..degree(p1) repeat
         Lin:L UP(x,R):=append(Lin:L UP(x,R),[0]:L UP(x,R))
   Lin

-- Computation of the subresultant sequence Sres(j)(P,p,Q,q) when:
-- deg(P) = p and deg(Q) = q and p > q
subresultantSequence(p1,p2):L UP(x,R) ==
   p:NNI:=degree(p1)
   q:NNI:=degree(p2)
   List1:L UP(x,R):=subresultantSequenceInner(p1,p2)
   List2:L UP(x,R):=[p1,p2]
   c1:R:=leadingCoefficient(p1)
   for j in 3..#(List1) repeat
      Pr0:UP(x,R):=List1.j
      Pr1:UP(x,R):=(Pr0 exquo c1**((p-q-1)::NNI))::UP(x,R)
      List2:L UP(x,R):=append(List2:L UP(x,R),[Pr1]:L UP(x,R))
   List2

-- Computation of the sign (+1,0,-1) of an element in an ordered integral
delta(int1:NNI):R ==
   (-1)**((int1*(int1+1) exquo 2)::NNI)

-- Computation of the delta function:
delta(int1:NNI):R ==
   (-1)**((int1*(int1+1) exquo 2)::NNI)

-- Computation of the Sturm-Habicht sequence of two polynomials P and Q
-- in R[x] where R is an ordered integral domain
polsth1(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
   sc1:R:=(sign(c1))::R
Pr1:UP(x,R):=pseudoRemainder(differentiate(p1)*p2,p1)
Pr2:UP(x,R):=(Pr1 exquo c1**(q::NNI))::UP(x,R)
c2:R:=leadingCoefficient(Pr2)
r:NNI:=degree(Pr2)
Pr3:UP(x,R):=monomial(sc1**(p-r-1::NNI),0)*p1
Pr4:UP(x,R):=monomial(sc1**(p-r-1::NNI),0)*Pr2
Listf:L UP(x,R):=[Pr3,Pr4]
if r < p-1 then
  Pr5:UP(x,R):=monomial(delta((p-r-1)::NNI)*c2**((p-r-1)::NNI),0)*Pr2
  for j in ((r+1)::INT)..((p-2)::INT) repeat
    Listf:L UP(x,R):=append(Listf:L UP(x,R),[0]:L UP(x,R))
    Listf:L UP(x,R):=append(Listf:L UP(x,R),[Pr5]:L UP(x,R))
if Pr1=0 then List1:L UP(x,R):=Listf
else List1:L UP(x,R):=subresultantSequence(p1,Pr2)
List2:L UP(x,R):=[]
for j in 0..((r-1)::INT) repeat
  Pr6:UP(x,R):=monomial(delta((p-j-1)::NNI),0)*List1.((p-j+1)::NNI)
  List2:L UP(x,R):=append([Pr6]:L UP(x,R),List2:L UP(x,R))
append(Listf:L UP(x,R),List2:L UP(x,R))

polsth2(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
sc1:R:=(sign(c1))::R
Pr1:UP(x,R):=monomial(sc1,0)*p1
Pr2:UP(x,R):=differentiate(p1)*p2
Pr3:UP(x,R):=monomial(sc1,0)*Pr2
Listf:L UP(x,R):=[Pr1,Pr3]
List1:L UP(x,R):=subresultantSequence(p1,Pr2)
List2:L UP(x,R):=[]
for j in 0..((p-2)::INT) repeat
  Pr2:UP(x,R):=monomial(delta((v-j)::NNI),0)*List1.((v-j+1)::NNI)
  Pr3:UP(x,R):=(Pr2 exquo c1)::UP(x,R)
  List2:L UP(x,R):=append([Pr3]:L UP(x,R),List2:L UP(x,R))
append(Listf:L UP(x,R),List2:L UP(x,R))

polsth3(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
sc1:R:=(sign(c1))::R
q1:NNI:=(q-1)::NNI
v:NNI:=(p+q1)::NNI
Pr1:UP(x,R):=monomial(delta(q1::NNI)*sc1**((q+1)::NNI),0)*p1
Listf:L UP(x,R):=[Pr1]
List1:L UP(x,R):=subresultantSequence(differentiate(p1)*p2,p1)
List2:L UP(x,R):=[]
for j in 0..((p-1)::NNI) repeat
  Pr2:UP(x,R):=monomial(delta((v-j)::NNI),0)*List1.((v-j+1)::NNI)
  Pr3:UP(x,R):=(Pr2 exquo c1)::UP(x,R)
  List2:L UP(x,R):=append([Pr3]:L UP(x,R),List2:L UP(x,R))
append(Listf:L UP(x,R),List2:L UP(x,R))

SturmHabichtSequence(p1,p2):L UP(x,R) ==
p:NNI:=degree(p1)
q: NNI := degree(p2)
c1: R := leadingCoefficient(p1)
c1 = 1 or q = 1 => polynomial1(p1, p2, q, c1)
q = 0 => polynomial2(p1, p2, q, c1)
polynomial3(p1, p2, q, c1)

-- Computation of the Sturm-Habicht principal coefficients of two
-- polynomials P and Q in \( \mathbb{R}[x] \) where R is an ordered integral domain

SturmHabichtCoefficients(p1, p2): List R ==
  List1: List UP(x, R) := SturmHabichtSequence(p1, p2)
  List2: List R := []
  qp: NNI := #(List1) :: NNI
  [coefficient(p, (qp - j) :: NNI) for p in List1 for j in 1..qp]
  for j in 1..qp repeat
    Ply: R := coefficient(List1.j, (qp - j) :: NNI)
    List2 := append(List2, [Ply])
  List2

-- Computation of the number of sign variations of a list of non zero
-- elements in an ordered integral domain

variation(Lsig: List R): INT ==
  size?(Lsig, 1) => 0
  elt1: R := first Lsig
  elt2: R := Lsig.2
  sig1: R := (sign(elt1 * elt2)) :: R
  List1: List R := rest Lsig
  sig1 = 1 => variation List1
  1 + variation List1

-- Computation of the number of sign permanences of a list of non zero
-- elements in an ordered integral domain

permanence(Lsig: List R): INT ==
  size?(Lsig, 1) => 0
  elt1: R := first Lsig
  elt2: R := Lsig.2
  sig1: R := (sign(elt1 * elt2)) :: R
  List1: List R := rest Lsig
  sig1 = -1 => permanence List1
  1 + permanence List1

-- Computation of the functional \( W \) which works over a list of elements
-- in an ordered integral domain, with non zero first element
qzeros(Lsig:L R) :=
  while last Lsig = 0 repeat
    Lsig := reverse rest reverse Lsig
  Lsig

epsil(int1:NNI,elt1:R,elt2:R):INT ==
  int1 = 0 => 0
  odd? int1 => 0
  ct1:INT := if elt1 > 0 then 1 else -1
  ct2:INT := if elt2 > 0 then 1 else -1
  ct3:NNI := (int1 exquo 2)::NNI
  ct4:INT := (ct1*ct2)::INT
  ((-1)**(ct3::NNI))*ct4

numbnce(Lsig:L R):NNI ==
  null Lsig => 0
  eltp := Lsig.1
  eltp = 0 => 0
  1 + numbce(rest Lsig)

numbce(Lsig:L R):NNI ==
  null Lsig => 0
  eltp := Lsig.1
  not(eltp = 0) => 0
  1 + numbce(rest Lsig)

wfunctaux(Lsig:L R):INT ==
  null Lsig => 0
  List2 := []
  List1 := Lsig:
  cont1 := numbce(List1:
  for j in 1..cont1 repeat
    List2 := append(List2,[first List1]):
    List1 := rest List1
  ind2 := 0
  cont2 := numbce(List1:
  for j in 1..cont2 repeat
    List1 := rest List1
  ind2 := epsil(cont2::NNI, last List2, first List1)
  ind3 := permanence(List2::L R) - variation(List2::L R)
  ind4 := ind2 + ind3
  ind4 + wfunctaux(List1:

wfunct(Lsig:L R):INT ==
  List1 := Lsig:
  wfunctaux(List1:

-- Computation of the integer number:
-- #\{a in Rc(R)/P(a)=0 Q(a)>0\} - #\{a in Rc(R)/P(a)=0 Q(a)<0\}
-- where:
-- - \( R \) is an ordered integral domain,
-- - \( \text{Re}(R) \) is the real closure of \( R \),
-- - \( P \) and \( Q \) are polynomials in \( R[x] \),
-- - by \#\( A \) we note the cardinal of the set \( A \)

-- In particular:
-- - \( \text{SturmHabicht}(P,1) \) is the number of "real" roots of \( P \),
-- - \( \text{SturmHabicht}(P,Q^2) \) is the number of "real" roots of \( P \) making \( Q \) neq 0

\[
\text{SturmHabicht}(p1,p2) \text{::INT} ==
\]

\[
\text{print("+") :: Ex}
\]

\[
p2 = 0 \Rightarrow 0
degree(p1:UP(x,R)) = 0 \Rightarrow 0
\]

\[
\text{List1:L UP(x,R):=}\text{SturmHabichtSequence}(p1,p2)
\]

\[
\text{qp:NNI:=}(\#(\text{List1)::NNI})
\]

\[
\text{wfunct [coefficient(p,((qp-j)::NNI)) for p in List1 for j in 1..qp]}
\]

\[
\text{countRealRoots}(p1) \text{::INT} == \text{SturmHabicht}(p1,1)
\]

if \( R \) has \( \text{GcdDomain} \) then

\[
\text{SturmHabichtMultiple}(p1,p2) \text{::INT} ==
\]

\[
\text{print("+") :: Ex}
\]

\[
p2 = 0 \Rightarrow 0
degree(p1:UP(x,R)) = 0 \Rightarrow 0
\]

\[
\text{SH:L UP(x,R):=}\text{SturmHabichtSequence}(p1,p2)
\]

\[
\text{qp:NNI:=}(\#(\text{SH)::NNI})
\]

\[
\text{ans:= wfunct [coefficient(p,((qp-j)::NNI)) for p in SH for j in 1..qp]}
\]

\[
\text{SH:=reverse SH}
\]

\[
\text{while first SH = 0 repeat SH:=rest SH}
\]

\[
degree first SH = 0 \Rightarrow \text{ans}
\]

-- OK: it probably wasn’t square free, so this item is probably the
-- \( \text{gcd} \) of \( p1 \) and \( p1' \)
-- unless \( p1 \) and \( p2 \) have a factor in common (naughty!)
\[
differentiate(p1) \text{ exquo first SH case UP(x,R) =>}
\]

-- it was the \( \text{gcd} \) of \( p1 \) and \( p1' \)

\[
\text{ans+SturmHabichtMultiple(first SH,p2)}
\]

\[
\text{sqfr:=factorList squareFree p1}
\]

\[
\#sqfr = 1 \text{ and sqfr.first.xpnt=1 => ans}
\]

\[
\text{reduce("+",[f.xpnt*SturmHabicht(f.fctr,p2) for f in sqfr])}
\]

\[
\text{countRealRootsMultiple}(p1) \text{::INT} == \text{SturmHabichtMultiple}(p1,1)
\]

— SHP.dotabb —

"SHP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SHP"]
package SUBRESP SubResultantPackage

-- SubResultantPackage.input --

)set break resume
)sys rm -f SubResultantPackage.output
)spool SubResultantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SubResultantPackage
--E 1

)spool
)lisp (bye)

-- SubResultantPackage.help --

====================================================================
SubResultantPackage examples
====================================================================

This package computes the subresultants of two polynomials which is needed for the ‘Lazard Rioboo’ enhancement to Tragers integrations formula. For efficiency reasons this has been rewritten to call Lionel Ducos package which is currently the best one.

See Also:
  o )show SubResultantPackage
SubResulantPackage (SUBRESP)

Exports:

primitivePart    subresultantVector

— package SUBRESP SubResulantPackage —

)abbrev package SUBRESP SubResulantPackage
++ Author: Barry Trager, Renaud Rioboo
++ Date Created: 1987
++ Date Last Updated: August 2000
++ Description:
++ This package computes the subresultants of two polynomials which is needed
++ for the ‘Lazard Rioboo’ enhancement to Trager’s integrations formula
++ For efficiency reasons this has been rewritten to call Lionel Ducos
++ package which is currently the best one.

SubResulantPackage(R, UP): Exports == Implementation where
R : IntegralDomain
UP: UnivariatePolynomialCategory R

Z ==> Integer
N ==> NonNegativeInteger

Exports ==> with
  subresultantVector: (UP, UP) -> PrimitiveArray UP
    ++ where pi is the i-th subresultant of p and q.
    ++ In particular, \spad{p0 = resultant(p, q)}.

if R has EuclideanDomain then
  primitivePart : (UP, R) -> UP
    ++ primitivePart(p, q) reduces the coefficient of p
    ++ modulo q, takes the primitive part of the result,
    ++ and ensures that the leading coefficient of that
    ++ result is monic.

Implementation ==> add
Lionel ==> PseudoRemainderSequence(R,UP)

if R has EuclideanDomain then
  primitivePart(p, q) ==
    rec := extendedEuclidean(leadingCoefficient p, q, 1)::Record(coef1:R, coef2:R)
    unitCanonical primitivePart map(x1 +-> (rec.coef1 * x1) rem q, p)

subresultantVector(p1, p2) ==
  F : UP -- auxiliary stuff !
  res : PrimitiveArray(UP) := new(2+max(degree(p1),degree(p2)), 0)
  --
  -- kind of stupid interface to Lionel's Package !!!!!!!!!!!!!
  -- might have been wiser to rewrite the loop ...
  -- But I'm too lazy. [rr]
  --
  l := chainSubResultants(p1,p2)$Lionel
  --
  -- this returns the chain of non null subresultants !
  -- we must rebuild subresultants from this.
  -- we really hope Lionel Ducos minded what he wrote
  -- since we are fully blind !
  --
  null l =>
    -- Hum it seems that Lionel returns [] when min(|p1|,|p2|) = 0
    zero?(degree(p1)) =>
      res.degree(p2) := p2
      if degree(p2) > 0 then
        res.((degree(p2)-1)::NonNegativeInteger) := p1
        res.0 := (leadingCoefficient(p1)**(degree p2)) :: UP
      else
        -- both are of degree 0 the resultant is 1 according to Loos
        res.0 := 1
      res
    zero?(degree(p2)) =>
      if degree(p1) > 0 then
        res.((degree(p1)-1)::NonNegativeInteger) := p2
        res.0 := (leadingCoefficient(p2)**(degree p1)) :: UP
      else
        -- both are of degree 0 the resultant is 1 according to Loos
        res.0 := 1
      res
  error "SUBRESP: strange Subresultant chain from PRS"
  Sn := first(l)
  --
  -- as of Loos definitions last subresultant should not be defective
l := rest(l)
n := degree(Sn)
F := Sn
null l => error "SUBRESP: strange Subresultant chain from PRS"
zero? Sn => error "SUBRESP: strange Subresultant chain from PRS"
while (l ^= []) repeat
  res.(n) := Sn
  F := first(l)
l := rest(l)
  -- F is potentially defective
  if degree(F) = n
    then
    -- F is defective
    -- null l => error "SUBRESP: strange Subresultant chain from PRS"
    Sn := first(l)
l := rest(l)
n := degree(Sn)
    res.((n-1)::NonNegativeInteger) := F
  else
    -- F is non defective
    -- degree(F) < n => error "strange result !"
    Sn := F
    n := degree(Sn)
    -- Lionel forgets about p1 if |p1| > |p2|
    -- forgets about p2 if |p2| > |p1|
    -- but he reminds p2 if |p1| = |p2|
    -- a glance at Loos should correct this !
    -- res.n := Sn
    -- -- Loos definition
    -- if degree(p1) = degree(p2)
      then
        res.((degree p1)+1) := p1
      else
        if degree(p1) > degree(p2)
          then
            res.(degree p1) := p1
          else
            res.(degree p2) := p2
        res
package SUPFRACF SupFractionFactorizer

SupFractionFactorize contains the factor function for univariate polynomials over the quotient field of a ring S such that the package MultivariateFactorize works for S

See Also:
  o )show SupFractionFactorizer
SupFractionFactorizer (SUPFRACF)

Exports:
factor    squareFree

--- package SUPFRACF SupFractionFactorizer ---

)abbrev package SUPFRACF SupFractionFactorizer
++ Author: P. Gianni
++ Date Created: October 1993
++ Date Last Updated: March 1995
++ Description:
++ SupFractionFactorizer contains the factor function for univariate
++ polynomials over the quotient field of a ring S such that the package
++ MultivariateFactorize works for S

SupFractionFactorizer(E,OV,R,P) : C == T
where
  E : OrderedAbelianMonoidSup
  OV : OrderedSet
  R : GcdDomain
  P : PolynomialCategory(R,E,OV)
  FP ==> Fraction P
  SUP ==> SparseUnivariatePolynomial

C == with
  factor : SUP FP -> Factored SUP FP
    + factor(p) factors the univariate polynomial p with coefficients
    + which are fractions of polynomials over R.
  squareFree : SUP FP -> Factored SUP FP
    + squareFree(p) returns the square-free factorization of the univariate polynomial p with coefficients
    + which are fractions of polynomials over R. Each factor has no repeated roots and the factors are
    + pairwise relatively prime.

T == add
  MFACT ==> MultivariateFactorize(OV,E,R,P)
  MSQFR ==> MultivariateSquareFree(E,OV,R,P)
UPCF2 => UnivariatePolynomialCategoryFunctions2

factor(p: SUP FP) : Factored SUP FP ==
  p=0 => 0
  R has CharacteristicZero and R has EuclideanDomain =>
    pden : P := lcm [denom c for c in coefficients p]
    pol : SUP FP := (pdens::FP)*p
    ipol: SUP P := map(numer, pol)$UPCF2(FP, SUP FP, P, SUP P)
    ffact: Factored SUP P := 0
    ffact := factor(ipol)$MFACT
    makeFR((1/pden * map(coerce, unit ffact)$UPCF2(P, SUP P, FP, SUP FP)),
    [["prime", map(coerce, u.factor)$UPCF2(P, SUP P, FP, SUP FP),
      u.exponent] for u in factors ffact])

squareFree p

squareFree(p: SUP FP) : Factored SUP FP ==
  p=0 => 0
  pden : P := lcm [denom c for c in coefficients p]
  pol : SUP FP := (pdens::FP)*p
  ipol: SUP P := map(numer, pol)$UPCF2(FP, SUP FP, P, SUP P)
  ffact: Factored SUP P := 0
  if R has CharacteristicZero and R has EuclideanDomain then
    ffact := squareFree(ipol)$MSQFR
  else ffact := squareFree(ipol)
  makeFR((1/pden * map(coerce, unit ffact)$UPCF2(P, SUP P, FP, SUP FP)),
  [["sqfr", map(coerce, u.factor)$UPCF2(P, SUP P, FP, SUP FP),
    u.exponent] for u in factors ffact])

— SUPFRACF.dotabb —

"SUPFRACF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUPFRACF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUPFRACF" -> "PFECAT"

package ODESYS SystemODESolver

— SystemODESolver.input —

)set break resume
)sys rm -f SystemODESolver.output
SystemODESolver (ODESYS)

Exports:
solve  solveInField  triangulate
package ODESYS SystemODESolver

)abbrev package ODESYS SystemODESolver
++ Author: Manuel Bronstein
++ Date Created: 11 June 1991
++ Date Last Updated: 13 April 1994
++ Description:
++ SystemODESolver provides tools for triangulating
++ and solving some systems of linear ordinary differential equations.

SystemODESolver(F, LO):Exports == Implementation where
   F: Field
   LO: LinearOrdinaryDifferentialOperatorCategory F

   N ==> NonNegativeInteger
   Z ==> Integer
   MF ==> Matrix F
   M ==> Matrix LO
   V ==> Vector F
   UF ==> Union(F, "failed")
   UV ==> Union(V, "failed")
   REC ==> Record(mat: M, vec: V)
   FSL ==> Record(particular: UF, basis: List F)
   VSL ==> Record(particular: UV, basis: List V)
   SOL ==> Record(particular: F, basis: List F)
   USL ==> Union(SOL, "failed")
   ER ==> Record(C: MF, g: V, eq: LO, rh: F)

Exports ==> with
   triangulate: (MF, V) -> Record(A:MF, eqs: List ER)
   triangulate: (M, V) -> REC
   solve: (MF, V, (LO, F) -> USL) -> Union(Record(particular: V, basis: MF), "failed")
   solveInField: (M, V, (LO, F) -> FSL) -> VSL
++ solveInField(m, v, solve) returns \spad{[[v_1,...,v_m], v_p]} such
++ that the solutions in \spad{F} of the system \spad{m x = v} are
++ \spad{v_p + c_1 v_1 + ... + c_m v_m} where the \spad{c_i's} are
++ constants, and the \spad{v_i's} form a basis for the solutions of
++ \spad{m x = 0}.
++ Argument \spad{solve} is a function for solving a single linear
++ ordinary differential equation in \spad{F}.

Implementation ==> add

import PseudoLinearNormalForm F

applyLodo : (M, Z, V, N) -> F
applyLodo0 : (M, Z, Matrix F, Z, N) -> F
backsolve : (M, V, (LO, F) -> FSL) -> VSL
firstnonzero: (M, Z) -> Z
FSL2USL : FSL -> USL
M2F : M -> Union(MF, "failed")

diff := D()$LO

solve(mm, v, solve) ==
rec := triangulate(mm, v)
sols:List(SOL) := empty()
for e in rec.eqs repeat
(u := solve(e.eq, e.rh)) case "failed" => return "failed"
sols := concat(u::SOL, sols)
n := nrows(rec.A) -- dimension of original vectorspace
k:N := 0 -- sum of sizes of visited companionblocks
i:N := 0 -- number of companionblocks
m:N := 0 -- number of Solutions
part:V := new(n, 0)
-- count first the different solutions
for sol in sols repeat
m := m + count((f1:F):Boolean +-> f1 ^= 0, sol.basis)$List(F)
SolMatrix:MF := new(n, m, 0)
m := 0
for sol in reverse_! sols repeat
i := i+1
er := rec.eqs.i
nn := #(er.g) -- size of active companionblock
for s in sol.basis repeat
solVec:V := new(n, 0)
-- compute corresponding solution base with recursion (24)
solVec(k+1) := s
for l in 2..nn repeat solVec(k+l) := diff solVec(k+l-1)
m := m+1
setColumn!(SolMatrix, m, solVec)
-- compute with (24) the corresponding components of the part. sol.
part(k+1) := sol.particular
for l in 2..nn repeat part(k+l) := diff part(k+l-1) - (er.g)(l-1)
k := k + n

-- transform these values back to the original system

triangulate(m:MF, v:V) ==
k:N := 0 -- sum of companion-dimensions
rat := normalForm(m, 1, (f1:F):F +-> - diff f1)
l := companionBlocks(rat.R, rat.Ainv * v)
ler:List(ER) := empty()
for er in l repeat
  n := nrows(er.C) -- dimension of this companion vectorspace
  op:LO := 0 -- compute homogeneous equation
  for j in 0..n-1 repeat op := op + monomial((er.C)(n, j + 1), j)
  op := monomial(1, n) - op
  sum:V := new(n::N, 0) -- compute inhomogen Vector (25)
  for j in 1..n-1 repeat sum(j+1) := diff(sum j) + (er.g) j
  h0:F := 0 -- compute inhomogenity (26)
  for j in 1..n repeat h0 := h0 - (er.C)(n, j) * sum j
  h0 := h0 + diff(sum n) + (er.g) n
  ler := concat([er.C, er.g, op, h0], ler)
k := k + n
[rat.A, ler]

-- like solveInField, but expects a system already triangularized
backsolve(m, v, solve) ==
part:V
r := maxRowIndex m
offset := minIndex v - (mr := minRowIndex m)
while r >= mr and every?(zero?, row(m, r))$Vector(LO) repeat r := r - 1
r < mr => error "backsolve: system has a 0 matrix"
(c := firstnonzero(m, r)) ^= maxColIndex m =>
  error "backsolve: undetermined system"
rec := solve(m(r, c), v(r + offset))
dim := (r - mr + 1)::N
if (part? := ((u := rec.particular) case F)) then
  part := new(dim, 0) -- particular solution
  part(r + offset) := u::F
-- hom is the basis for the homogeneous solutions, each column is a solution
hom:Matrix(F) := new(dim, #(rec.basis), 0)
for i in minColIndex hom .. maxColIndex hom for b in rec.basis repeat
  hom(r, i) := b
n:N := 1 -- number of equations already solved
while r > mr repeat
  r := r - 1
  c := c - 1
  (c := firstnonzero(m, r)) ^= c => error "backsolve: undetermined system"
  degree(eq := m(r, c)) > 0 => error "backsolve: pivot of order > 0"
a := leadingCoefficient(eq)::F
if part? then
  part(r + offset) := (v(r + offset) - applyLodo(m, r, part, n)) / a
for i in minColIndex hom .. maxColIndex hom repeat
hom(r, i) := - applyLodo0(m, r, hom, i, n)
n := n + 1
bas:List(V) := [column(hom,i) for i in minColIndex hom..maxColIndex hom]
part? => [part, bas]
["failed", bas]
solveInField(m, v, solve) ==
((n := nrows m) = ncols m) and
((u := M2F(diagonalMatrix [diff for i in 1..n] - m)) case MF) =>
(uu := solve(u::MF, v,
(l1:L0,f2:F):USL +-> FSL2USL solve(l1, f2))) case "failed" =>
["failed", empty()]
rc := uu::Record(particular:V, basis:MF)
[rc.particular, [column(rc.basis, i) for i in 1..ncols(rc.basis)]]
rec := triangulate(m, v)
backsolve(rec.mat, rec.vec, solve)
M2F m ==
mf:MF := new(nrows m, ncols m, 0)
for i in minRowIndex m .. maxRowIndex m repeat
  for j in minColIndex m .. maxColIndex m repeat
    (u := retractIfCan(m(i, j))@Union(F, "failed")) case "failed" =>
      return "failed"
    mf(i, j) := u::F
  mf
FSL2USL rec ==
rec.particular case "failed" => "failed"
[rec.particular::F, rec.basis]
-- returns the index of the first nonzero entry in row r of m
firstnonzero(m, r) ==
  for c in minColIndex m .. maxColIndex m repeat
    m(r, c) ^= 0 => return c
  error "firstnonzero: zero row"
-- computes +/[m(r, i) v(i) for i ranging over the last n columns of m]
applyLodo(m, r, v, n) ==
ans:F := 0
  c := maxColIndex m
  cv := maxIndex v
  for i in 1..n repeat
    ans := ans + m(r, c) (v cv)
    c := c - 1
    cv := cv - 1
  ans
-- computes +/[m(r, i) mm(i, c) for i ranging over the last n columns of m]
applyLodo0(m, r, mm, c, n) ==

ans := 0
rr := maxRowIndex mm
cc := maxColIndex m
for i in 1..n repeat
   ans := ans + m(r, cc) mm(rr, c)
   cc := cc - 1
   rr := rr - 1
ans

triangulate(m:M, v:V) ==
x := copy m
w := copy v
nrows := maxRowIndex x
ncols := maxColIndex x
minr := i := minRowIndex x
offset := minIndex w - minr
for j in minColIndex x .. ncols repeat
   if i > nrows then leave x
   rown := minr - 1
   for k in i .. nrows repeat
      if (x(k, j) ^= 0) and ((rown = minr - 1) or
         degree x(k,j) < degree x(rown,j)) then rown := k
   rown = minr - 1 => "enuf"
x := swapRows_!(x, i, rown)
swap_!(w, i + offset, rown + offset)
for k in i+1 .. nrows | x(k, j) ^= 0 repeat
   l := rightLcm(x(i,j), x(k,j))
a := rightQuotient(l, x(i, j))
b := rightQuotient(l, x(k, j))
-- l = a x(i,j) = b x(k,j)
for k1 in j+1 .. ncols repeat
   x(k, k1) := a * x(i, k1) - b * x(k, k1)
x(k, j) := 0
w(k + offset) := a(w(i + offset)) - b(w(k + offset))
i := i+1
[x, w]
package SYSSOLP SystemSolvePackage

--- SystemSolvePackage.input ---

)set break resume
)sys rm -f SystemSolvePackage.output
)spool SystemSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SystemSolvePackage
--E 1

)spool
)lisp (bye)

--- SystemSolvePackage.help ---

====================================================================
SystemSolvePackage examples
====================================================================

Symbolic solver for systems of rational functions with coefficients in an integral domain R.

The systems are solved in the field of rational functions over R. Solutions are exact of the form variable = value when the value is a member of the coefficient domain R. Otherwise the solutions are implicitly expressed as roots of univariate polynomial equations over R.

Care is taken to guarantee that the denominators of the input equations do not vanish on the solution sets.

The arguments to solve can either be given as equations or as rational functions interpreted as equal to zero. The user can specify an explicit list of symbols to be solved for, treating all other symbols appearing as parameters or omit the list of symbols in which case the system tries to solve with respect to all symbols appearing in the input.

See Also:
o )show SystemSolvePackage
SystemSolvePackage (SYSSOLP)

Exports:
solve triangularSystems

--- package SYSSOLP SystemSolvePackage ---

)abbrev package SYSSOLP SystemSolvePackage
++ Author: P. Gianni
++ Date Created: summer 1988
++ Date Last Updated: summer 1990
++ Description:
++ Symbolic solver for systems of rational functions with coefficients
++ in an integral domain R.
++ The systems are solved in the field of rational functions over R.
++ Solutions are exact of the form variable = value when the value is
++ a member of the coefficient domain R. Otherwise the solutions
++ are implicitly expressed as roots of univariate polynomial equations over R.
++ Care is taken to guarantee that the denominators of the input
++ equations do not vanish on the solution sets.
++ The arguments to solve can either be given as equations or
++ as rational functions interpreted as equal
++ to zero. The user can specify an explicit list of symbols to
++ be solved for, treating all other symbols appearing as parameters
++ or omit the list of symbols in which case the system tries to
++ solve with respect to all symbols appearing in the input.

SystemSolvePackage(R): Cat == Cap where
  NNI   ==> NonNegativeInteger
  P     ==> Polynomial
  EQ    ==> Equation
L ==> List
V ==> Vector
M ==> Matrix
UP ==> SparseUnivariatePolynomial
SE ==> Symbol
IE ==> IndexedExponents Symbol
SUP ==> SparseUnivariatePolynomial
R : IntegralDomain
F ==> Fraction Polynomial R
PP2 ==> PolynomialFunctions2(R,F)
PPR ==> Polynomial Polynomial R

Cat == with
solve: (L F, L SE) -> L L EQ F
  ++ solve(lp,lv) finds the solutions of the list lp of
  ++ rational functions with respect to the list of symbols lv.
solve: (L EQ F, L SE) -> L L EQ F
  ++ solve(le,lv) finds the solutions of the
  ++ list le of equations of rational functions
  ++ with respect to the list of symbols lv.
solve: L F -> L L EQ F
  ++ solve(lp) finds the solutions of the list lp of rational
  ++ functions with respect to all symbols appearing in lp.
solve: L EQ F -> L L EQ F
  ++ solve(le) finds the solutions of the list le of equations of
  ++ rational functions with respect to all symbols appearing in le.
solve: (F, SE) -> L EQ F
  ++ solve(p,v) solves the equation p=0, where p is a rational function
  ++ with respect to the variable v.
solve: (EQ F,SE) -> L EQ F
  ++ solve(eq,v) finds the solutions of the equation
  ++ eq with respect to the variable v.
solve: F -> L EQ F
  ++ solve(p) finds the solution of a rational function p = 0
  ++ with respect to the unique variable appearing in p.
solve: EQ F -> L EQ F
  ++ solve(eq) finds the solutions of the equation eq
  ++ with respect to the unique variable appearing in eq.

triangularSystems: (L F, L SE) -> L L P R
  ++ triangularSystems(lf,lv) solves the system of equations
  ++ defined by lf with respect to the list of symbols lv;
  ++ the system of equations is obtaining
++ by equating to zero the list of rational functions lf.
++ The output is a list of solutions where
++ each solution is expressed as a "reduced" triangular system of
++ polynomials.

Cap == add

import MPolyCatRationalFunctionFactorizer(IE,SE,R,P F)

---- Local Functions ----
linsolve: (L F, L SE) -> Union(L EQ F, "failed")
makePolys : L EQ F -> L F
makeR2F(r : R) : F == r :: (P R) :: F
makeP2F(p:P F):F ==
   lv:=variables p
   lv = [] => retract p
   for v in lv repeat p:=pushdown(p,v)
   retract p
makeEq(p:P F,lv:L SE): EQ F ==
   z:=last lv
   np:=numer makeP2F p
   lx:=variables np
   for x in lv repeat if member?(x,lx) then leave x
   up:=univariate(np,x)
   (degree up)=1 =>
      equation(x::P(R)::F,-coefficient(up,0)/leadingCoefficient up)
   equation(np::F,0$F)
varInF(v: SE): F == v::P(R) :: F
newInF(n: Integer):F==varInF new()$SE
testDegree(f :P R , lv :L SE) : Boolean ==
   "or"/[degree(f,vv)>0 for vv in lv]  

---- Exported Functions ----
-- solve a system of rational functions
triangularSystems(lf: L F,lv:L SE) : L L P R ==
   empty? lv => empty()
   empty? lf => empty()
   #lf = 1 =>
      p:= numer(first lf)
      fp:=(factor p)$GeneralizedMultivariateFactorize(SE,IE,R,R,P R)
      [[ff.factor] for ff in factors fp | testDegree(ff.factor,lv)]
      dmp:=DistributedMultivariatePolynomial(lv,P R)
      OV:=OrderedVariableList(lv)
      DP:=DirectProduct(#lv, NonNegativeInteger)
push:=PushVariables(R,DP,OV,dmp)
lq : L dmp
lvv:L OV:=[variable(vv)::OV for vv in lv]
lq:=[pushup(df::dmp,lvv)$push for f in lf|(df:=denom f)^=1]
lp:=[pushup(numer(f)::dmp,lvv)$push for f in lf]
parRes:=groebSolve(lp,lvv)$GroebnerSolve(lv,P R,R)
if lq=[] then
gb:=GroebnerInternalPackage(P R,DirectProduct(#lv,NNI),OV,dmp)
parRes:=[pr for pr in parRes |
and/
[redPol(fq,pr pretend List(dmp))$gb]^=0
for fq in lq]
[[retract pushdown(pf,lvv)$push for pf in pr] for pr in parRes]

-- One polynomial. Implicit variable --
solve(pol : F) ==
zero? pol =>
error "equation is always satisfied"
lv:=removeDuplicates
concat(variables numer pol, variables denom pol)
empty? lv => error "inconsistent equation"
#lv>1 => error "too many variables"
solve(pol,first lv)

-- general solver. Input in equation style. Implicit variables --
solve(eq : EQ F) ==
pol:= lhs eq - rhs eq
zero? pol =>
error "equation is always satisfied"
lv:=removeDuplicates
concat(variables numer pol, variables denom pol)
empty? lv => error "inconsistent equation"
#lv>1 => error "too many variables"
solve(pol,first lv)

-- general solver. Input in equation style --
solve(eq:EQ F,var:SE) == solve(lhs eq - rhs eq,var)

-- general solver. Input in polynomial style --
solve(pol:F,var:SE) ==
if R has GcdDomain then
    p:=primitivePart(numer pol,var)
    fp:=(factor p)$GeneralizedMultivariateFactorize(SE,IE,R,R,P R)
    [makeEq(map(makeR2F,ff.factor)$PP2,[var]) for ff in factors fp]
else empty()

-- Convert a list of Equations in a list of Polynomials
makePolys(l: L EQ F):L F == [lhs e - rhs e for e in l]

-- linear systems solver. Input as list of polynomials --
linSolve(lp:L F,lv:L SE) ==
CHAPTER 20. CHAPTER S

rec := Record(particular: Union(V F, "failed"), basis: L V F)

lr := L P R := [numer f for f in lp]

rec := linSolve(lr, lv) \(\text{LinearSystemPolynomialPackage}(R, \text{IE}, \text{SE}, \text{PR})\)
rec.particular case "failed" => "failed"
rhs := rec.particular :: V F
zeron := V F := zero(#lv)

for p in rec.basis | p ^= zeron repeat
  sym := newInF(1)
  for i in 1..#lv repeat
    rhs.i := rhs.i + sym*p.i

eqs := L EQ F := []
for i in 1..#lv repeat
  eqs := append(eqs, [(lv.i)::(PR)::F = rhs.i])
eqs

--- general solver. Input in polynomial style. Implicit variables ---
solve(lr : L F) ==
  lv := "setUnion"/[setUnion(variables numer p, variables denom p)
  for p in lr]
  solve(lr, lv)

--- general solver. Input in equation style. Implicit variables ---
solve(le : L EQ F) ==
  lr := makePolys le
  lv := "setUnion"/[setUnion(variables numer p, variables denom p)
  for p in lr]
  solve(lr, lv)

--- general solver. Input in equation style ---
solve(le : L EQ F, lv : L SE) == solve(makePolys le, lv)

checkLinear(lr : L F, vl : L SE) : Boolean ==
  ld := [denom pol for pol in lr]
  for f in ld repeat
    if (or/[member?(x, vl) for x in variables f]) then return false
    and/[totalDegree(numer pol, vl) < 2 for pol in lr]

--- general solver. Input in polynomial style ---
solve(lr : L F, vl : L SE) ==
  empty? vl => empty()
  checkLinear(lr, vl) =>
    -- linear system --
    soln := linSolve(lr, vl)
    soln case "failed" => []
eqs := L EQ F := []
  for i in 1..#vl repeat
    lhs := (vl.i)::(PR)::F
    rhs := rhs soln.i
    eqns := append(eqns, [lhs = rhs])
  [eqns]
-- polynomial system --
if R has GcdDomain then
parRes:=triangularSystems(lr,vl)
[[makeEq(map(makeR2F,f)$FP2,vl) for f in pr]
  for pr in parRes]
else [[]]

---
SYSSOLP.dotabb ---
"SYSSOLP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SYSSOLP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SYSSOLP" -> "ALIST"

package SGCF SymmetricGroupCombinatoricFunctions

--- SymmetricGroupCombinatoricFunctions.input ---

)set break resume
)sys rm -f SymmetricGroupCombinatoricFunctions.output
)spool SymmetricGroupCombinatoricFunctions.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show SymmetricGroupCombinatoricFunctions
-- E 1

)spool
)lisp (bye)

---
SymmetricGroupCombinatoricFunctions.help ---

====================================================================
SymmetricGroupCombinatoricFunctions examples
====================================================================
SymmetricGroupCombinatoricFunctions contains combinatoric functions concerning symmetric groups and representation theory: list young tableaus, improper partitions, subsets bijection of Coleman.

See Also:
  o )show SymmetricGroupCombinatoricFunctions

---

SymmetricGroupCombinatoricFunctions (SGCF)

Exports:
  colem
  inverseColeman
  listYoungTableaus
  makeYoungTableau
  nextColeman
  nextLatticePermutation
  nextPartition
  numberOfImproperPartitions
  subSet
  unrankImproperPartitions0
  unrankImproperPartitions1

— package SGCF SymmetricGroupCombinatoricFunctions —

)abbrev package SGCF SymmetricGroupCombinatoricFunctions
++ Authors: Johannes Grabmeier, Thorsten Werther
++ Date Created: 03 September 1988
++ Date Last Updated: 07 June 1990
++ References:
  ++ G. James/ A. Kerber: The Representation Theory of the Symmetric
  ++ S.G. Williamson: Combinatorics for Computer Science,
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Description:
++ SymmetricGroupCombinatoricFunctions contains combinatoric
++ functions concerning symmetric groups and representation
++ theory: list young tableaus, improper partitions, subsets
++ bijection of Coleman.

SymmetricGroupCombinatoricFunctions(): public == private where

   NNI ==> NonNegativeInteger
   I ==> Integer
   L ==> List
   M ==> Matrix
   V ==> Vector
   B ==> Boolean
   ICF ==> IntegerCombinatoricFunctions Integer

public ==> with

-- IS THERE A WORKING DOMAIN Tableau ??
-- coerce : M I -> Tableau(I)
--   ++ coerce(ytab) coerces the Young-Tableau ytab to an element of
--   ++ the domain Tableau(I).

coleman : (L I, L I, L I) -> M I
   ++ coleman(alpha,beta,pi):
   ++ there is a bijection from the set of matrices having nonnegative
   ++ entries and row sums alpha, column sums beta
   ++ to the set of Salpha - Sbeta double cosets of the
   ++ symmetric group Sn. (Salpha is the Young subgroup
   ++ corresponding to the improper partition alpha).
   ++ For a representing element pi of such a double coset,
   ++ coleman(alpha,beta,pi) generates the Coleman-matrix
   ++ corresponding to alpha, beta, pi.
   ++ Note that The permutation pi of \{1,2,...,n\} has to be given
   ++ in list form.
   ++ Note that the inverse of this map is inverseColeman
   ++ (if pi is the lexicographical smallest permutation
   ++ in the coset). For details see James/Kerber.
inverseColeman : (L I, L I, M I) -> L I
   ++ inverseColeman(alpha,beta,C):
   ++ there is a bijection from the set of matrices having nonnegative
   ++ entries and row sums alpha, column sums beta
   ++ to the set of Salpha - Sbeta double cosets of the
   ++ symmetric group Sn. (Salpha is the Young subgroup
   ++ corresponding to the improper partition alpha).
   ++ For such a matrix C, inverseColeman(alpha,beta,C)
++ calculates the lexicographical smallest \( \pi \) in the
++ corresponding double coset.
++ Note that the resulting permutation \( \pi \) of \( \{1,2,\ldots,n\} \)
++ is given in list form.
++ Notes: the inverse of this map is coleman.
++ For details, see James/Kerber.

\text{listYoungTableaus} : (L \ I) \rightarrow L \ M \ I
++ \text{listYoungTableaus}(\lambda) \text{ where } \lambda \text{ is a proper partition}
++ generates the list of all standard tableaus of shape \( \lambda \)
++ by means of lattice permutations. The numbers of the lattice
++ permutation are interpreted as column labels. Hence the
++ contents of these lattice permutations are the conjugate of
++ \( \lambda \).
++ Notes: the functions \text{nextLatticePermutation} and
++ \text{makeYoungTableau} are used.
++ The entries are from 0,\ldots,n-1.

\text{makeYoungTableau} : (L \ I, L \ I) \rightarrow M \ I
++ \text{makeYoungTableau}(\lambda, g) \text{ computes for a given lattice}
++ permutation \( g \) and for an improper partition \( \lambda \)
++ the corresponding standard tableau of shape \( \lambda \).
++ Notes: see \text{listYoungTableaus}.
++ The entries are from 0,\ldots,n-1.

\text{nextColeman} : (L \ I, L \ I, M \ I) \rightarrow M \ I
++ \text{nextColeman}(\alpha, \beta, C) \text{ generates the next Coleman matrix}
++ of column sums \( \alpha \) and row sums \( \beta \) according
++ to the lexicographical order from bottom-to-top.
++ The first Coleman matrix is achieved by \( C = \text{new}(1,1,0) \).
++ Also, \( \text{new}(1,1,0) \) indicates that \( C \) is the last Coleman matrix.

\text{nextLatticePermutation} : (L \ I, L \ I, B) \rightarrow L \ I
++ \text{nextLatticePermutation}(\lambda, l, \text{constructNotFirst}) \text{ generates}
++ the lattice permutation according to the proper partition
++ \( \lambda \) succeeding the lattice permutation \( l \) in
++ lexicographical order as long as \text{constructNotFirst} is true.
++ If \text{constructNotFirst} is false, the first lattice permutation
++ is returned.
++ The result \text{nil} indicates that \( l \) has no successor.

\text{nextPartition} : (V \ I, V \ I, I) \rightarrow V \ I
++ \text{nextPartition}(\gamma, \text{part}, \text{number}) \text{ generates the partition of}
++ \text{number which follows part according to the right-to-left}
++ lexicographical order. The partition has the property that
++ its components do not exceed the corresponding components of
++ \( \gamma \). The first partition is achieved by \text{part}=[].
++ Also, \( [] \) indicates that part is the last partition.

\text{nextPartition} : (L \ I, V \ I, I) \rightarrow V \ I
++ \text{nextPartition}(\gamma, \text{part}, \text{number}) \text{ generates the partition of}
++ \text{number which follows part according to the right-to-left}
++ lexicographical order. The partition has the property that
++ its components do not exceed the corresponding components of
++ \( \gamma \). The first partition is achieved by \text{part}=[].
++ Also, \( [] \) indicates that part is the last partition.
numberOfImproperPartitions: (I,I) -> I
  ++ numberOfImproperPartitions(n,m) computes the number of partitions
  ++ of the nonnegative integer n in m nonnegative parts with regarding
  ++ the order (improper partitions).
  ++ Example: numberOfImproperPartitions (3,3) is 10,
  ++ since [0,0,3], [0,1,2], [0,2,1], [0,3,0], [1,0,2], [1,1,1],
  ++ [1,2,0], [2,0,1], [2,1,0], [3,0,0] are the possibilities.
  ++ Note that this operation has a recursive implementation.

subSet : (I,I,I) -> L I
  ++ subSet(n,m,k) calculates the k-th m-subset of the set
  ++ 0,1,...,(n-1) in the lexicographic order considered as
  ++ a decreasing map from 0,...,(m-1) into 0,...,(n-1).
  ++ See S.G. Williamson: Theorem 1.60.
  ++ Error: if not (0 <= m <= n and 0 <= k <= (n choose m)).

unrankImproperPartitions0 : (I,I,I) -> L I
  ++ unrankImproperPartitions0(n,m,k) computes the k-th improper
  ++ partition of nonnegative n in m nonnegative parts in reverse
  ++ lexicographical order.
  ++ Example: [0,0,3] < [0,1,2] < [0,2,1] < [0,3,0] <
  ++ [1,0,2] < [1,1,1] < [1,2,0] < [2,0,1] < [2,1,0] < [3,0,0].
  ++ Error: if k is negative or too big.
  ++ Note that counting of subtrees is done by
  ++ numberOfImproperPartitions

unrankImproperPartitions1: (I,I,I) -> L I
  ++ unrankImproperPartitions1(n,m,k) computes the k-th improper
  ++ partition of nonnegative n in at most m nonnegative parts
  ++ ordered as follows: first, in reverse
  ++ lexicographically according to their non-zero parts, then
  ++ according to their positions (i.e. lexicographical order
  ++ using subSet: [3,0,0] < [0,3,0] < [0,0,3] < [2,1,0] <
  ++ [2,0,1] < [0,2,1] < [1,2,0] < [1,0,2] < [0,1,2] < [1,1,1].
  ++ Note that counting of subtrees is done by
  ++ numberOfImproperPartitionsInternal.

private == add

import Set I

-- declaration of local functions

numberOfImproperPartitionsInternal: (I,I,I) -> I
  -- this is used as subtree counting function in
  -- "unrankImproperPartitions1". For (n,m,cm) it counts
  -- the following set of m-tuples: The first (from left
  -- to right) m-cm non-zero entries are equal, the remaining
  -- positions sum up to n. Example: (3,3,2) counts
  -- [x,3,0], [x,0,3], [0,x,3], [x,2,1], [x,1,2], x non-zero.
-- definition of local functions

numberOfImproperPartitionsInternal(n,m,cm) ==
  n = 0 => binomial(m,cm)$ICF
  cm = 0 and n > 0 => 0
  s := 0
  for i in 0..n-1 repeat
    s := s + numberOfImproperPartitionsInternal(i,m,cm-1)
s
-- definition of exported functions

numberOfImproperPartitions(n,m) ==
  if n < 0 or m < 1 then return 0
  if m = 1 or n = 0 then return 1
  s := 0
  for i in 0..n repeat
    s := s + numberOfImproperPartitions(n-i,m-1)
s
unrankImproperPartitions0(n,m,k) ==
  l : L I := nil$(L I)
  k < 0 => error"counting of partitions is started at 0"
  k >= numberOfImproperPartitions(n,m) =>
    error"there are not so many partitions"
  for t in 0..(m-2) repeat
    s : I := 0
    for y in 0..n repeat
      sOld := s
      s := s + numberOfImproperPartitions(n-y,m-t-1)
      if s > k then leave
    l := append(l,list(y)$(L I))$(L I)
    k := k - sOld
    n := n - y
  l

unrankImproperPartitions1(n,m,k) ==
  -- we use the counting procedure of the leaves in a tree
  -- having the following structure: First of all non-zero
  -- labels for the sons. If addition along a path gives n,
  -- then we go on creating the subtree for (n choose cm)
  -- where cm is the length of the path. These subsets determine
  -- the positions for the non-zero labels for the partition
  -- to be formeded. The remaining positions are filled by zeros.
  nonZeros : L I := nil$(L I)
partition : V I := new(m::NNI,0$I)$(V I)
k < 0 => nonZeros
k >= numberOfImproperPartitions(n,m) => nonZeros
cm : I := m --cm gives the depth of the tree
while n ^= 0 repeat
  s : I := 0
cm := cm - 1
  for y in n..1 by -1 repeat --determination of the next son
    sOld := s -- remember old s
    s := s + numberOfImproperPartitionsInternal(n-y,m,cm)
    if s > k then leave
    -- y is the next son, so put it into the pathlist "nonZero"
    nonZeros := append(nonZeros,list(y)$(L I))$(L I)
    k := k - sOld --updating
    n := n - y --updating
  --having found all m-cm non-zero entries we change the structure
  --of the tree and determine the non-zero positions
  nonZeroPos : L I := reverse subSet(m,m-cm,k)
  --building the partition
  for i in 1..m-cm repeat partition.(1+nonZeroPos.i) := nonZeros.i
entries partition

subSet(n,m,k) ==
k < 0 or n < 0 or m < 0 or m > n =>
  error "improper argument to subSet"
bin : I := binomial$ICF (n,m)
k >= bin =>
  error "there are not so many subsets"
l : L I := []
n = 0 => l
mm : I := k
s : I := m
for t in 0..(m-1) repeat
  for y in (s-1)..(n+1) repeat
    if binomial$ICF (y,s) > mm then leave
  l := append (l,list(y-1)$(L I))
  mm := mm - binomial$ICF (y-1,s)
s := s-1
l

nextLatticePermutation(lambda, lattP, constructNotFirst) ==
lprime : L I := conjugate(lambda)$PartitionsAndPermutations
columns : NNI := (first(lambda)$(L I))::NNI
rows : NNI := (first(lprime)$(L I))::NNI
n : NNI :=(+/lambda):NNI
not constructNotFirst => -- first lattice permutation
  lattP := nil$(L I)
  for i in columns..1 by -1 repeat
    for l in 1..lprime(i) repeat
      lattP := cons(i,lattP)
    lattP

help : V I := new(columns,0) -- entry help(i) stores the number
-- of occurrences of number i on our way from right to left
rightPosition : NNI := n
leftEntry : NNI := lattP(rightPosition)::NNI
ready : B := false
until (ready or (not constructNotFirst)) repeat
  rightEntry : NNI := leftEntry
  leftEntry := lattP(rightPosition-1)::NNI
  help(rightEntry) := help(rightEntry) + 1
  -- search backward decreasing neighbour elements
  if rightEntry > leftEntry then
    if ((lprime(leftEntry)-help(leftEntry)) >
      (lprime(rightEntry)-help(rightEntry)+1)) then
      -- the elements may be swapped because the number of occurances
      -- of leftEntry would still be greater than those of rightEntry
      ready := true
      j : NNI := leftEntry + 1
      -- search among the numbers leftEntry+1...rightEntry for the
      -- smallest one which can take the place of leftEntry.
      -- negation of condition above:
      while (help(j)=0) or ((lprime(leftEntry)-lprime(j))
        < (help(leftEntry)-help(j)+2)) repeat j := j + 1
      lattP(rightPosition-1) := j
      help(j) := help(j)-1
      help(leftEntry) := help(leftEntry) + 1
      -- reconstruct the rest of the list in increasing order
      for l in rightPosition..n repeat
        j := 0
        while help(1+j) = 0 repeat j := j + 1
        lattP(1::NNI) := j+1
        help(1+j) := help(1+j) - 1
      -- end of "if rightEntry > leftEntry"
    rightPosition := (rightPosition-1)::NNI
    if rightPosition = 1 then constructNotFirst := false
    -- end of repeat-loop
  not constructNotFirst => nil$(L I)
lattP

makeYoungTableau(lambda,gitter) ==
lprime : L I := conjugate(lambda)$PartitionsAndPermutations
columns : NNI := (first(lambda)$(L I))::NNI
rows : NNI := (first(lprime)$(L I))::NNI
ytab : M I := new(rows,columns,0)
help : V I := new(columns,1)
i : I := -1  -- this makes the entries ranging from 0,...,n-1
    -- i := 0 would make it from 1,...,n.
j : I := 0
for l in 1..maxIndex gitter repeat
    j := gitter(l)
i := i + 1
    ytab(help(j),j) := i
    help(j) := help(j) + 1
ytab

-- coerce(ytab) ==
-- lli := listOfLists(ytab)$(M I)
-- -- remove the filling zeros in each row. It is assumed that
-- -- that there are no such in row 0.
-- for i in 2..maxIndex lli repeat
-- THIS IS DEFINITIVELY WRONG, I NEED A FUNCTION WHICH DELETES THE
-- 0s, in my version there are no mapping facilities yet.
-- deleteInPlace(not zero?,lli i)
-- tableau(lli)$Tableau(I)

listYoungTableaus(lambda) ==
    lattice : L I
    ytab : M I
    younglist : L M I := nil$(L M I)
    lattice := nextLatticePermutation(lambda,lattice,false)
    until null lattice repeat
        ytab := makeYoungTableau(lambda,lattice)
        younglist := append(younglist,[ytab]$(L M I))$(L M I)
        lattice := nextLatticePermutation(lambda,lattice,true)
    younglist

nextColeman(alpha,beta,C) ==
    nrow : NNI := #beta
    ncol : NNI := #alpha
    vnull : V I := vector(nil()$(L I)) -- empty vector
    vzero : V I := new(ncol,0)
    vrest : V I := new(ncol,0)
    cnull : M I := new(1,1,0)
    coleman := copy C
    if coleman ^= cnull then
        -- look for the first row of "coleman" that has a succeeding
        -- partition, this can be atmost row nrow-1
        i : NNI := (nrow-1)::NNI
        vrest := row(coleman,i) + row(coleman,nrow)
        --for k in 1..ncol repeat
vrest(k) := coleman(i,k) + coleman(nrow,k)
succ := nextPartition(vrest, row(coleman, i), beta(i))
while (succ = vnull) repeat
    if i = 1 then return cnull -- part is last partition
    i := (i - 1)::NNI
    for k in 1..ncol repeat
        vrest(k) := vrest(k) + coleman(i,k)
        vrest := vrest + row(coleman, i)
        succ := nextPartition(vrest, row(coleman, i), beta(i))
    j : I := i
    coleman := setRow!(coleman, i, succ)
    for k in 1..ncol repeat
        vrest(k) := vrest(k) - coleman(i,k)
        vrest := vrest - row(coleman, i)
    else
        vrest := vector alpha
        for k in 1..ncol repeat
            vrest(k) := alpha(k)
        coleman := new(nrow, ncol, 0)
        j : I := 0
        for i in (j+1)::NNI..nrow-1 repeat
            succ := nextPartition(vrest, vnull, beta(i))
            coleman := setRow!(coleman, i, succ)
            vrest := vrest - succ
            for k in 1..ncol repeat
                vrest(k) := vrest(k) - succ(k)
            setRow!(coleman, nrow, vrest)

nextPartition(gamma: V I, part: V I, number: I) ==
    nextPartition(entries gamma, part, number)

nextPartition(gamma: L I, part: V I, number: I) ==
    n : NNI := #gamma
    vnull : V I := vector(nil()$(L I)) -- empty vector
    if part = vnull then
        i : NNI := 2
        sum := part(1)
        while (part(i) = gamma(i)) or (sum = 0) repeat
            sum := sum + part(i)
            i := i + 1
        if i = 1+n then return vnull -- part is last partition
        sum := sum - 1
        part(i) := part(i) + 1
    else
        sum := number
        part := new(n, 0)
        i := 1+n
        j : NNI := 1
while sum > gamma(j) repeat
    part(j) := gamma(j)
    sum := sum - gamma(j)
    j := j + 1
    part(j) := sum
for k in j+1..i-1 repeat
    part(k) := 0

inverseColeman(alpha, beta, C) ==
    pi : L I := nil$(L I)
    nrow : NNI := #beta
    ncol : NNI := #alpha
    help : V I := new(nrow, 0)
    sum : I := 1
    for i in 1..nrow repeat
        help(i) := sum
        sum := sum + beta(i)
    for j in 1..ncol repeat
        for i in 1..nrow repeat
            for k in 2..1+C(i,j) repeat
                pi := append(pi, list(help(i))$(L I))
                help(i) := help(i) + 1
    pi

coleman(alpha, beta, pi) ==
    nrow : NNI := #beta
    ncol : NNI := #alpha
    temp : L L I := nil$(L L I)
    help : L I := nil$(L I)
    colematrix : M I := new(nrow, ncol, 0)
    betasum : NNI := 0
    alphasum : NNI := 0
    for i in 1..ncol repeat
        help := nil$(L I)
        for j in alpha(i)..1 by-1 repeat
            help := cons(pi(j::NNI+alphasum), help)
            alphasum := (alphasum + alpha(i))::NNI
        temp := append(temp, list(help)$(L L I))
    for i in 1..nrow repeat
        help := nil$(L I)
        for j in beta(i)..1 by-1 repeat
            help := cons(j::NNI+betasum, help)
            betasum := (betasum + beta(i))::NNI
        colematrix(i, j) := #intersect(brace(help), brace(temp(j)))
    colematrix
package SYMFUNC SymmetricFunctions

--- SymmetricFunctions.input ---

)set break resume
)sys rm -f SymmetricFunctions.output
)spool SymmetricFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SymmetricFunctions
--E 1

)spool
)lisp (bye)

--- SymmetricFunctions.help ---

====================================================================
SymmetricFunctions examples
====================================================================

Computes all the symmetric functions in n variables.

See Also:
  o )show SymmetricFunctions
SymmetricFunctions (SYMFUNC)

Exports:

- symFunc

-- package SYMFUNC SymmetricFunctions --

)abbrev package SYMFUNC SymmetricFunctions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 28 Jun 1990
++ Description:
++ Computes all the symmetric functions in n variables.

SymmetricFunctions(R:Ring): Exports == Implementation where
  UP ==> SparseUnivariatePolynomial R

Exports ==> with
  symFunc: List R -> Vector R
    ++ symFunc([r1,...,rn]) returns the vector of the
    ++ elementary symmetric functions in the \spad{ri's}:
    ++ \spad{[r1 + ... + rn, r1 r2 + ... + r(n-1) rn, ..., r1 r2 ... rn]}.
  symFunc: (R, PositiveInteger) -> Vector R
    ++ symFunc(r, n) returns the vector of the elementary
    ++ symmetric functions in \spad{[r,r,...,r]} \spad{n} times.

Implementation ==> add
  signFix: (UP, NonNegativeInteger) -> Vector R
  symFunc(x, n) == signFix((monomial(1, 1)$UP - x::UP) ** n, 1 + n)
  symFunc 1 ==
    signFix(*/[monomial(1, 1)$UP - a::UP for a in l], 1 + #l)
  signFix(p, n) ==
    m := minIndex(v := vectorise(p, n)) + 1
    for i in 0..((#v quo 2) - 1)::NonNegativeInteger repeat
\begin{verbatim}
qsetelt_!(v, 2*i + m, - qelt(v, 2*i + m))
reverse_! v
\end{verbatim}

— SYMFUNC.dotabb —

"SYMFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SYMFUNC"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"SYMFUNC" -> "IVECTOR"

—
Chapter 21

Chapter T

package TABLBUMP TableauxBumpers

— TableauxBumpers.input —

)set break resume
)sys rm -f TableauxBumpers.output
)spool TableauxBumpers.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TableauxBumpers
--E 1

)spool
)lisp (bye)

— TableauxBumpers.help —

====================================================================
TableauxBumpers examples
====================================================================

TableauBumpers implements the Schenstead-Knuth correspondence between sequences and pairs of Young tableaux. The 2 Young tableaux are represented as a single tableau with pairs as components.
See Also:
- )show TableauxBumpers

### TableauxBumpers (TABLBUMP)

Exports:
- bat
- bat1
- bumprow
- bumpatab
- bumpatab1
- inverse
- lex
- maxrow
- mr
- slex
- tab
- tab1
- untab

---

<table>
<thead>
<tr>
<th>package TABLBUMP TableauxBumpers</th>
</tr>
</thead>
</table>

)abbrev package TABLBUMP TableauxBumpers
++ Author: William H. Burge
++ Date Created: 1987
++ Date Last Updated: 23 Sept 1991
++ Description:
++ TableauxBumpers implements the Schenstead-Knuth
++ correspondence between sequences and pairs of Young tableaux.
++ The 2 Young tableaux are represented as a single tableau with
++ pairs as components.

TableauxBumpers(S:OrderedSet):T==C where
L==>List
ST==>Stream
B==>Boolean
ROW==>Record(fs:B,sd:L S,td:L L S)
RC==>Record(f1:L S,f2:L L S,f3:L S,f4:L L L S)
PAIR==>L S
T== with
  bumprow:((S,S)->B,PAIR,L PAIR)->ROW
  ++ bumprow(cf,pr,r) is an auxiliary function which
++ bumps a row r with a pair pr
++ using comparison function cf, and returns a record
bumpab:((S,S)->B,PAIR,L L PAIR)->L L PAIR
++ bumpab(cf,pr,t) bumps a tableau t with a pair pr
++ using comparison function cf, returning a new tableau
bumpab1:(PAIR,L L PAIR)->L L PAIR
++ bumpab1(pr,t) bumps a tableau t with a pair pr
++ using comparison function \spadfun{<},
++ returning a new tableau
untab: (L PAIR,L L PAIR)->L PAIR
++ untab(lp,llp) is an auxiliary function
++ which unbumps a tableau llp,
++ using lp to accumulate pairs
bat1:L PAIR->L PAIR
++ bat1(llp) unbumps a tableau llp.
++ Operation bat1 is the inverse of tab1.
bat:Tableau(L S)->L L S
++ bat(ls) unbumps a tableau ls
tab1:L PAIR->L L PAIR
++ tab1(lp) creates a tableau from a list of pairs lp
tab:L S->Tableau(L S)
++ tab(ls) creates a tableau from ls by first creating
++ a list of pairs using slex,
++ then creating a tableau using tab1.
lex:L PAIR->L PAIR
++ lex(ls) sorts a list of pairs to lexicographic order
slex:L S->L PAIR
++ slex(ls) sorts the argument sequence ls, then zips (see map) the
++ original argument sequence with the sorted result to
++ a list of pairs
inverse:L S->L S
++ inverse(ls) forms the inverse of a sequence ls
maxrow:(PAIR,L L PAIR,L PAIR,L L PAIR,L L PAIR,L L PAIR)->RC
++ maxrow(a,b,c,d,e) is an auxiliary function for mr
mr:L PAIR->RC
++ mr(t) is an auxiliary function which
++ finds the position of the maximum element of a tableau t
++ which is in the lowest row, producing a record of results
C== add
  cf:(S,S)->B
  bumprow(cf,x:(PAIR),lls:(L PAIR))==
    if null lls
    then [false,x,[x]]$ROW
    else (y:(PAIR):=first lls;
      if cf(x.2,y.2)
      then [true,[x.1,y.2],cons([y.1,x.2],rest lls)]$ROW
      else (rw:ROW:=bumprow(cf,x,rest lls);
        [rw.fs,rw.sd,cons(first lls,rw.td)]$ROW ))
bumpab(cf,x:(PAIR),llls:(L L PAIR))==
if null llls
then [[]]
else (rw:ROW:= bumptrow(cf,x,first llls);
    if rw.fs
    then cons(rw.td, bumptab(cf,rw.sd,rest llls))
    else cons(rw.td,rest llls))

bumptabi(x, llls)== bumptab((s1,s2) → s1<s2, x, llls)

rd==> reduce$StreamFunctions2(PAIR,L L PAIR)
tab1(llls:(L PAIR))== rd([],bumptabi,llls:(ST PAIR))

srt==>sort$(PAIR)
lexorder:(PAIR,PAIR)→B
lexorder(p1,p2)==if p1.1=p2.1 then p1.2<p2.2 else p1.1<p2.1
lex lp=(sort$(L PAIR))((s1,s2) → lexorder(s1,s2), lp)
slex ls=lex([[i,j] for i in srt((s1,s2) → s1<s2, ls) for j in ls])
inverse ls=[[s2.s for s1,s2 in
             lex([[j,i] for i in srt((s1,s2) → s1<s2, ls)
                  for j in ls])]

tab(lls:(PAIR))===(tableau tab1 slex ls )

maxrow(n,a,b,c,d,llls)==
    if null llls or null(first llls)
    then [n,a,b,c]$RC
    else (fst:=first first llls;rst:=rest first llls;
         if fst.1>n.1
         then maxrow(fst,d,rst,rest llls,cons(first llls,d),rest llls)
         else maxrow(n,a,b,c,cons(first llls,d),rest llls))

mr llls==maxrow(first first llls, [], rest first llls, rest llls,
                  [], llls)

untab(lp, llls)==
    if null llls
    then lp
    else (rc:RC:=mr llls;
        rv:=reverse (bumptab((s1:S,s2:S):B → s2<s1, rc. f1, rc. f2));
        untab(cons(first first rv,lp)
              ,append(rest rv,
                     if null rc.f3
                     then []
                     else cons(rc.f3,rc.f4))))

bat1 llls==untab([], [reverse lls for lls in llls])
bat tb==bat1(listOfLists tb)
--- TABLBUMP.dotabb ---

"TABLBUMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TABLBUMP"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"TABLBUMP" -> "FLAGG-"
"TABLBUMP" -> "FLAGG"

---

package TBCMPPK TabulatedComputationPackage

--- TabulatedComputationPackage.input ---

)set break resume
)sys rm -f TabulatedComputationPackage.output
)spool TabulatedComputationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TabulatedComputationPackage
--E 1

)spool
)lisp (bye)

---

--- TabulatedComputationPackage.help ---

====================================================================
TabulatedComputationPackage examples
====================================================================

TabulatedComputationPackage(Key ,Entry) provides some modest support
for dealing with operations with type Key -> Entry. The result of
such operations can be stored and retrieved with this package by using
a hash-table. The user does not need to worry about the management of
this hash-table. However, only one hash-table is built by calling
TabulatedComputationPackage(Key ,Entry).

See Also:
(show TabulatedComputationPackage

---

TabulatedComputationPackage (TBCMPPK)

---

Exports:
clearTable! extractIfCan initTable! insert! makingStats?
printInfo! printStats! printingInfo? startStats! usingTable?

— package TBCMPPK TabulatedComputationPackage —

)abbrev package TBCMPPK TabulatedComputationPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/09/1998
++ Date Last Updated: 12/16/1998
++ Description:
++ \axiom{TabulatedComputationPackage(Key ,Entry)} provides some modest support
++ for dealing with operations with type \axiom{Key -> Entry}. The result of
++ such operations can be stored and retrieved with this package by using
++ a hash-table. The user does not need to worry about the management of
++ this hash-table. However, only one hash-table is built by calling
++ \axiom{TabulatedComputationPackage(Key ,Entry)}.

TabulatedComputationPackage(Key ,Entry): Exports == Implementation where
Key: SetCategory
Entry: SetCategory
N ==> NonNegativeInteger
H ==> HashTable(Key, Entry, "UEQUAL")
iprintpack ==> InternalPrintPackage()

Exports == with
  initTable!: () -> Void
    ++ \axiom{initTable!()} initializes the hash-table.
printInfo!: (String, String) -> Void
++ \text{printInfo!(x,y)} initializes the messages to be printed
++ when manipulating items from the hash-table. If
++ a key is retrieved then \text{x} is displayed. If an item is
++ stored then \text{y} is displayed.

startStats!: (String) -> Void
++ \text{startStats!(x)} initializes the statistics process and
++ sets the comments to display when statistics are printed

printStats!: () -> Void
++ \text{printStats!()} prints the statistics.

clearTable!: () -> Void
++ \text{clearTable!()} clears the hash-table and assumes that
++ it will no longer be used.

usingTable?: () -> Boolean
++ \text{usingTable?()} returns true iff the hash-table is used

printingInfo?: () -> Boolean
++ \text{printingInfo?()} returns true iff messages are printed
++ when manipulating items from the hash-table.

makingStats?: () -> Boolean
++ \text{makingStats?()} returns true iff the statistics process
++ is running.

extractIfCan: Key -> Union(Entry,"failed")
++ \text{extractIfCan(x)} searches the item whose key is \text{x}.

insert!: (Key, Entry) -> Void
++ \text{insert!(x,y)} stores the item whose key is \text{x} and whose
++ entry is \text{y}.

Implementation == add

initTable!(): Void ==
  table? := true
  t := empty()
  void()

printStats!(s1: String, s2: String): Void ==
  (empty? s1) or (empty? s2) => void()
  not usingTable? =>
  error "in printStats!(s1,s2) of TBCMPPK: not allowed to use hashtable"
  info? := true
  ok := s1
  ko := s2
  void()

startStats!(s: String): Void ==
empty? s => void()
not table? =>
    error "in startStats!()$TBCMPPK: not allowed to use hashtable"
stats? := true
used := 0
domainName := s
void()
printStats!(): Void ==
not table? =>
    error "in printStats!()$TBCMPPK: not allowed to use hashtable"
not stats? =>
    error "in printStats!()$TBCMPPK: statistics not started"
output(" ")$OutputPackage
title: String := concat("*** ", concat(domainName," Statistics ***"))
output(title)$OutputPackage
n: N := #t
output(" Table size: ", n::OutputForm)$OutputPackage
output(" Entries reused: ", used::OutputForm)$OutputPackage
clearTable!(): Void ==
not table? =>
    error "in clearTable!()$TBCMPPK: not allowed to use hashtable"
t := empty()
table? := false
info? := false
stats? := false
domainName := empty()$String
void()
usingTable?() == table?
printingInfo?() == info?
makingStats?() == stats?
extractIfCan(k: Key): Union(Entry,"failed") ==
not table? => "failed" :: Union(Entry,"failed")
s: Union(Entry,"failed") := search(k,t)
s case Entry =>
    if info? then iprint(ok)$iprintpack
    if stats? then used := used + 1
    return s
"failed" :: Union(Entry,"failed")
insert!(k: Key, e:Entry): Void ==
not table? => void()
t.k := e
if info? then iprint(ko)$iprintpack
void()
package TANEXP TangentExpansions

--- TangentExpansions.input ---

)set break resume
)sys rm -f TangentExpansions.output
)spool TangentExpansions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TangentExpansions
--E 1

)spool
)lisp (bye)

---

--- TangentExpansions.help ---

====================================================================
TangentExpansions examples
====================================================================

Expands tangents of sums and scalar products.

See Also:
  o )show TangentExpansions

---
TangentExpansions (TANEXP)

Exports:
   tanAn  tanNa  tanSum

--- package TANEXP TangentExpansions ---

)abbrev package TANEXP TangentExpansions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 20 Apr 1990
++ Description:
++ Expands tangents of sums and scalar products.

TangentExpansions(R:Field): Exports == Implementation where
   PI ==> PositiveInteger
   Z  ==> Integer
   UP ==> SparseUnivariatePolynomial R
   QF ==> Fraction UP

Exports ==> with
   tanSum: List R -> R
       ++ tanSum([a1,...,an]) returns \spad{f(a1,...,an)} such that
       ++ if \spad{ai = tan(ui)} then \spad{f(a1,...,an) = tan(u1 + ... + un)}.
   tanAn : (R, PI) -> UP
       ++ tanAn(a, n) returns \spad{P(x)} such that
       ++ if \spad{a = tan(u)} then \spad{P(tan(u/n)) = 0}.
   tanNa : (R, Z) -> R
       ++ tanNa(a, n) returns \spad{f(a)} such that
       ++ if \spad{a = tan(u)} then \spad{f(a) = tan(n * u)}.

Implementation ==> add
   import SymmetricFunctions(R)
   import SymmetricFunctions(UP)

   mtoN : Integer -> Integer
   tanPIa: PI -> QF
\begin{verbatim}
mitoN n == (odd? n => -1; 1)
tanAn(a, n) == a * denom(q := tanPIa n) - numer q

tanNa(a, n) ==
  zero? n => 0
  negative? n => -tanNa(a, -n)
  (numer(t := tanPIa(n::PI)) a) / ((denom t) a)

  tanSum l ==
    m := minIndex(v := symFunc l)
    +/[mitoN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
    / +/[mitoN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]

-- tanPIa(n) returns P(a)/Q(a) such that
-- if a = tan(u) then P(a)/Q(a) = tan(n * u);
tanPIa n ==
  m := minIndex(v := symFunc(monomial(1, 1)$UP, n))
  +/[mitoN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
  / +/[mitoN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]

package UTSSOL TaylorSolve

  -- TaylorSolve.input --

)set break resume
)sys rm -f TaylorSolve.output
)spool TaylorSolve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TaylorSolve
\end{verbatim}
--E 1

)spool
)lisp (bye)

---

— TaylorSolve.help —

====================================================================
TaylorSolve examples
====================================================================

UTSSOL is a facility to compute the first few coefficients of a Taylor
series given only implicitly by a function f that vanishes when applied to
the Taylor series.

It uses the method of undetermined coefficients.

Could I either
take a function UTSCAT F $\rightarrow$ UTSCAT F and still be able to compute
with undetermined coefficients, or
take a function F $\rightarrow$ F, and do likewise?

Let’s see.

Try to compute the equation without resorting to power series. I.e., \%
c: SUP SUP F, and f: SUP SUP F $\rightarrow$ SUP SUP F. Won’t this make the
computation of coefficients terribly slow?

I could also try to replace transcendental kernels with variables\dots

Unfortunately, SUP F does not have TRANFUN -- well, it can’t, of
course. However, I’d like to be able to compute \%
\sin(1+\text{monomial}(1,1)$UFPS SUP EXPR INT).

See Also:
o )show TaylorSolve

---
TaylorSolve (UTSSOL)

Exports:
seriesSolve

--- package UTSSOL TaylorSolve ---

)abbrev package UTSSOL TaylorSolve
++ Description:
++ This package has no description

TaylorSolve(F, UTSF, UTSSUPF): Exports == Implementation where
  F: Field
  SUP ==> SparseUnivariatePolynomialExpressions
  UTSF: UnivariateTaylorSeriesCategory F
  UTSSUPF: UnivariateTaylorSeriesCategory SUP F
  NNI ==> NonNegativeInteger

Exports == with
  seriesSolve: (UTSSUPF -> UTSSUPF, List F) -> UTSF

Implementation == add

\getchunk{implementation: UTSSOL TaylorSolve}

--- implementation: UTSSOL TaylorSolve ---

seriesSolve(f, l) ==
c1 :=
  map((x:F):SUP F +-> x::(SUP F), l)
  $ListFunctions2(F, SUP F)::(Stream SUP F)
coeffs: Stream SUP F := concat(c1, generate(monomial(1$F,1$NNI)))

---
coeffs is the stream of the already computed coefficients of the solution, plus one which is so far undetermined. We store in st.2 the complete stream and in st.1 the stream starting with the first coefficient that has possibly not yet been computed.

The mathematics is not quite worked out. If coeffs is initialized as a stream with all coefficients set to the \emph{same} transcendental value, and not enough initial values are given, then the missing ones are implicitly assumed to be all identical. It may well happen that a solution is produced, although it is not uniquely determined.

— implementation: UTSSOL TaylorSolve —

\begin{verbatim}
  st: List Stream SUP F := [coeffs, coeffs]
\end{verbatim}

Consider an arbitrary equation $f(x, y(x)) = 0$. When setting $x = 0$, we obtain $f(0, y(0)) = 0$. It is not necessarily the case that this determines $y(0)$ uniquely, so we need one initial value that satisfies this equation.

seriesSolve should check that the given initial values satisfy $f(0, y(0), y'(0), ...) = 0$.

Now consider the derivatives of $f$, where we write $y$ instead of $y(x)$ for better readability:

$$\frac{d}{dx} f(x, y) = f_1(x, y) + f_2(x, y)y'$$

and

$$\frac{d^2}{dx^2} f(x, y) = f_{1,1}(x, y) + f_{1,2}(x, y)y' + f_{2,1}(x, y)y' + f_{2,2}(x, y)(y')^2 + f_2(x, y)y''.$$  

In general, $\frac{d^2}{dx^2} f(x, y)$ depends only linearly on $y''$.

This points to another possibility: Since we know that we only need to solve linear equations, we could compute two values and then use interpolation. This might be a bit slower, but more importantly: can we still check that we have enough initial values? Furthermore, we then really need that $f$ is analytic, i.e., operators are not necessarily allowed anymore. However, it seems that composition is allowed.

— implementation: UTSSOL TaylorSolve —
If the next element was already calculated, we can simply return it:

--- implementation: UTSSOL TaylorSolve ---

then

res := ground coeff
st.1 := rest nr

else

Otherwise, we have to find the first non-satisfied relation and solve it. It should be linear, or a single non-constant monomial. That is, the solution should be unique.

--- implementation: UTSSOL TaylorSolve ---

ns := st.2
eqs: Stream SUP F := coefficients f series ns
while zero? first eqs repeat eqs := rest eqs
eq: SUP F := first eqs
if degree eq > 1 then
    if monomial? eq then res := 0
else
    output(hconcat("The equation is: ", eq::OutputForm))$OutputPackage
    error "seriesSolve: equation for coefficient not linear"
else res := (-coefficient(eq, 0$NNI)$(SUP F)
    /coefficient(eq, 1$NNI)$(SUP F))

nr.1 := res::SUP F
--
concat!(st.2, monomial(1$F,1$NNI))
st.1 := rest nr

res
series generate next

---

--- UTSSOL.dotabb ---
package TEMUTL TemplateUtilities

--- TemplateUtilities.input ---

)set break resume
)sys rm -f TemplateUtilities.output
)spool TemplateUtilities.output
)set message test on
)set message auto off
)clear all

--$ 1 of 1
)show TemplateUtilities
--E 1

)spool
)lisp (bye)

---

--- TemplateUtilities.help ---

====================================================================
TemplateUtilities examples
====================================================================

This package provides functions for template manipulation

See Also:
  o )show TemplateUtilities

---
TemplateUtilities (TEMUTL)

Exports:

interpreterString  stripCommentsAndBlanks

— package TEMUTL TemplateUtilities —

)abbrev package TEMUTL TemplateUtilities
+ Author: Mike Dewar
++ Date Created: October 1992
++ Description:
++ This package provides functions for template manipulation

TemplateUtilities(): Exports == Implementation where

Exports == with

interpreterString : String -> Any
++ interpreterString(s) treats a string as a piece of AXIOM input, by
++ parsing and interpreting it.

stripCommentsAndBlanks : String -> String
++ stripCommentsAndBlanks(s) treats s as a piece of AXIOM input, and
++ removes comments, and leading and trailing blanks.

Implementation == add

import InputForm

stripC(s:String,u:String):String ==
i : Integer := position(u,s,1)
i = 0 => s
    delete(s,i..)

stripCommentsAndBlanks(s:String):String ==
    trim(stripC(stripC(s,"++"),"--"),char " ")

parse(s:String):InputForm ==
    ncParseFromString(s)$Lisp::InputForm
interpretString(s: String): Any ==
  interpret parse s

---

-- TEMUTL.dotabb --

"TEMUTL" [color="#FF4488", href="bookvol10.4.pdf#nameddest=TEMUTL"]
"STRING" [color="#88FF44", href="bookvol10.3.pdf#nameddest=STRING"]
"TEMUTL" -> "STRING"

---

package TEX1 TexFormat1

-- TexFormat1.input --

)set break resume
)sys rm -f TexFormat1.output
)spool TexFormat1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TexFormat1
--E 1

)spool
)lisp (bye)

---

-- TexFormat1.help --

====================================================================
TexFormat1 examples
====================================================================

TexFormat1 provides a utility coercion for changing to TeX format anything that has a coercion to the standard output format.
TexFormat1 (TEX1)

Exports:
coerce

— package TEX1 TexFormat1 —

)abbrev package TEX1 TexFormat1
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ Description:
++ \spadtype{TexFormat1} provides a utility coercion for changing
++ to TeX format anything that has a coercion to the standard output format.

TexFormat1(S : SetCategory): public == private where
public == with
  coerce: S -> TexFormat()
  ++ coerce(s) provides a direct coercion from a domain S to
  ++ TeX format. This allows the user to skip the step of first
  ++ manually coercing the object to standard output format before
  ++ it is coerced to TeX format.

private == add
import TexFormat()

coerce(s : S): TexFormat ==
  coerce(s :: OutputForm)$TexFormat
package TOOLSIGN ToolsForSign

— ToolsForSign.input —

)set break resume
)sys rm -f ToolsForSign.output
)spool ToolsForSign.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ToolsForSign
--E 1

)spool
)lisp (bye)

— ToolsForSign.help —

====================================================================
ToolsForSign examples
====================================================================

Tools for the sign finding utilities.

See Also:
  o )show ToolsForSign
ToolsForSign (TOOLSIGN)

Exports:
  direction nonQsign sign

— package TOOLSIGN ToolsForSign —

)abbrev package TOOLSIGN ToolsForSign
++ Author: Manuel Bronstein
++ Date Created: 25 August 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Tools for the sign finding utilities.

ToolsForSign(R:Ring): with
  sign : R -> Union(Integer, "failed")
  + sign(r) undocumented
  nonQsign : R -> Union(Integer, "failed")
  + nonQsign(r) undocumented
  direction: String -> Integer
  + direction(s) undocumented
== add

if R is AlgebraicNumber then
  nonQsign r ==
  sign((r pretend AlgebraicNumber)::Expression(Integer))$ElementaryFunctionSign(Integer, Expression Integer)
else
  nonQsign r == "failed"

if R has RetractableTo Fraction Integer then
  sign r ==
  (u := retractIfCan(r)@Union(Fraction Integer, "failed"))
  case Fraction(Integer) => sign(u::Fraction Integer)
\begin{verbatim}
nonQsign r

else
  if R has RetractableTo Integer then
    sign r ==
    (u := retractIfCan(r)@Union(Integer, "failed"))
    case "failed" => "failed"
    sign(u::Integer)
  else
    sign r ==
    zero? r => 0
    one? r => 1
    r = 1 => 1
    r = -1 => -1
    "failed"

direction st ==
  st = "right" => 1
  st = "left" => -1
  error "Unknown option"

— TOOLSIGN.dotabb —
"TOOLSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOOLSIGN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"TOOLSIGN" -> "FS"

package DRAW TopLevelDrawFunctions

— TopLevelDrawFunctions.input —

)set break resume
)sys rm -f TopLevelDrawFunctions.output
)spool TopLevelDrawFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
\end{verbatim}
TopLevelDrawFunctions examples

TopLevelDrawFunctions provides top level functions for drawing graphics of expressions.

See Also:
  o )show TopLevelDrawFunctions

Exports:
  draw    makeObject

---

--- package DRAW TopLevelDrawFunctions ---

)abbrev package DRAW TopLevelDrawFunctions
++ Author: Clifton J. Williamson
++ Date Created: 23 January 1990
TopLevelDrawFunctions(Ex:Join(ConvertibleTo InputForm,SetCategory)):
Exports == Implementation where
B => Boolean
BIND => SegmentBinding Float
L => List
SF => DoubleFloat
DROP => DrawOption
PPC => ParametricPlaneCurve Ex
PPCF => ParametricPlaneCurve(SF -> SF)
PSC => ParametricSpaceCurve Ex
PSCF => ParametricSpaceCurve(SF -> SF)
PSF => ParametricSurface Ex
PSFF => ParametricSurface(((SF,SF) -> SF)
SPACE3 => ThreeSpace(SF)
VIEW2 => TwoDimensionalViewport
VIEW3 => ThreeDimensionalViewport
Exports => with

--% Two Dimensional Function Plots

draw: (Ex,BIND,L DROP) -> VIEW2
draw: (Ex,BIND) -> VIEW2

--% Parametric Plane Curves
draw: (PPC,BIND,L DROP) -> VIEW2
draw: (PPC,BIND) -> VIEW2
--% Parametric Space Curves

\texttt{draw: (PSC,BIND,L DROP) -> VIEW3}
++ \texttt{draw(curve(f(t),g(t),h(t)),t = a..b,l) draws the graph of the}
++ parametric curve \texttt{spad(x = f(t)), spad(y = g(t)), spad(z = h(t))}
++ as \texttt{t ranges from spad(min(a,b)) to spad(max(a,b)); spad(h(t))}
++ is the default title, and the options contained in the list 1 of
++ the domain \texttt{spad(DrawOption)} are applied.

\texttt{draw: (PSC,BIND) -> VIEW3}
++ \texttt{draw(curve(f(t),g(t),h(t)),t = a..b) draws the graph of the parametric}
++ curve \texttt{spad(x = f(t)), spad(y = g(t)), spad(z = h(t)) as \texttt{t ranges}}
++ \texttt{from spad(min(a,b)) to spad(max(a,b)); spad(h(t)) is the default}
++ title.

\texttt{makeObject: (PSC,BIND,L DROP) -> SPACE3}
++ \texttt{makeObject(curve(f(t),g(t),h(t)),t = a..b,l) returns a space of}
++ the domain \texttt{spadtype(ThreeSpace)} which contains the graph of the
++ parametric curve \texttt{spad(x = f(t)), spad(y = g(t)), spad(z = h(t))}
++ as \texttt{t ranges from spad(min(a,b)) to spad(max(a,b)); spad(h(t))}
++ is the default title, and the options contained in the list 1 of
++ the domain \texttt{spad(DrawOption)} are applied.

\texttt{makeObject: (PSC,BIND) -> SPACE3}
++ \texttt{makeObject(curve(f(t),g(t),h(t)),t = a..b) returns a space of the}
++ domain \texttt{spadtype(ThreeSpace)} which contains the graph of the
++ parametric curve \texttt{spad(x = f(t)), spad(y = g(t)), spad(z = h(t))}
++ as \texttt{t ranges from spad(min(a,b)) to spad(max(a,b)); spad(h(t)) is}
++ the default title.

--% Three Dimensional Function Plots

\texttt{draw: (Ex,BIND,BIND,L DROP) -> VIEW3}
++ \texttt{draw(f(x,y),x = a..b,y = c..d,l) draws the graph of \texttt{spad(z = f(x,y)}}
++ as \texttt{x ranges from \texttt{spad(min(a,b)) to spad(max(a,b)) and y ranges from}
++ \texttt{spad(min(c,d)) to spad(max(c,d)); spad(f(x,y)) is the default}
++ title, and the options contained in the list 1 of the domain
++ \texttt{spad(DrawOption)} are applied.

\texttt{draw: (Ex,BIND,BIND) -> VIEW3}
++ \texttt{draw(f(x,y),x = a..b,y = c..d) draws the graph of \texttt{spad(z = f(x,y)}}
++ as \texttt{x ranges from \texttt{spad(min(a,b)) to spad(max(a,b)) and y ranges from}
++ \texttt{spad(min(c,d)) to spad(max(c,d)); spad(f(x,y)) appears in the title bar.}

\texttt{makeObject: (Ex,BIND,BIND,L DROP) -> SPACE3}
++ \texttt{makeObject(f(x,y),x = a..b,y = c..d,l) returns a space of the}
++ domain \texttt{spadtype(ThreeSpace)} which contains the graph of
++ \texttt{spad(z = f(x,y)) as \texttt{x ranges from \texttt{spad(min(a,b)) to spad(max(a,b))}
++ and \texttt{y ranges from \texttt{spad(min(c,d)) to spad(max(c,d)); spad(f(x,y))}
++ is the default title, and the options contained in the list 1 of the
++ domain \texttt{spad(DrawOption)} are applied.

\texttt{makeObject: (Ex,BIND,BIND) -> SPACE3}
++ \texttt{makeObject(f(x,y),x = a..b,y = c..d) returns a space of the domain}
++ \texttt{spadtype(ThreeSpace) which contains the graph of \texttt{spad(z = f(x,y)}}
++ as \texttt{x ranges from \texttt{spad(min(a,b)) to spad(max(a,b)) and y ranges from}
++ \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{f(x,y)} appears as the ++ default title.

--% Parametric Surfaces

\texttt{draw: (PSF,BIND,BIND,L DROP) -> VIEW3}
++ \texttt{draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) draws the ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)}, ++ \spad{z = h(u,v)} as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)} ++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{h(t)} ++ is the default title, and the options contained in the list l of ++ the domain \spad{\text{DrawOption}} are applied.}

\texttt{draw: (PSF,BIND,BIND) -> VIEW3}
++ \texttt{draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) draws the ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)}, ++ \spad{z = h(u,v)} as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)} ++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{h(t)} is ++ the default title.}

\texttt{makeObject: (PSF,BIND,BIND,L DROP) -> SPACE3}
++ \texttt{makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) returns ++ a space of the domain \spad{\text{ThreeSpace}} which contains the graph ++ of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)}, ++ \spad{z = h(u,v)} as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)} ++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{h(t)} is ++ the default title, and the options contained in the list l of ++ the domain \spad{\text{DrawOption}} are applied.}

\texttt{makeObject: (PSF,BIND,BIND) -> SPACE3}
++ \texttt{makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) returns ++ a space of the domain \spad{\text{ThreeSpace}} which contains the ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)}, ++ \spad{z = h(u,v)} as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)} ++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{h(t)} is ++ the default title.}

Implementation ==> add
import TopLevelDrawFunctionsForCompiledFunctions
import MakeFloatCompiledFunction(Ex)
import ParametricPlaneCurve(SF -> SF)
import ParametricSpaceCurve(SF -> SF)
import ParametricSurface((SF,SF) -> SF)
import ThreeSpace(SF)

--% Two Dimensional Function Plots

\texttt{draw(f:Ex,bind:BIND,l:L DROP) ==}
-- create title if necessary
if not option?(l,"title" :: Symbol) then
s:String := unparse(convert(f)@InputForm)
if sayLength(s)$DisplayPackage > 50 then
  l := concat(title "AXIOM2D",l)
else l := concat(title s,l)
-- call 'draw'
draw(makeFloatFunction(f,variable bind),segment bind,l)

draw(f:Ex,bind:BIND) == draw(f,bind,nil())

--% Parametric Plane Curves
draw(ppc:PPC,bind:BIND,1:L DROP) ==
f := coordinate(ppc,1); g := coordinate(ppc,2)
-- create title if necessary
if not option?(l,"title" :: Symbol) then
s:String := unparse(convert(f)@InputForm)
if sayLength(s)$DisplayPackage > 50 then
  l := concat(title "AXIOM2D",l)
else l := concat(title s,l)
-- create curve with functions as coordinates
curve : PPCF := curve(makeFloatFunction(f,variable bind),_
  makeFloatFunction(g,variable bind))$PPCF
-- call 'draw'
draw(curve,segment bind,l)

draw(ppc:PPC,bind:BIND) == draw(ppc,bind,nil())

------------------------------------------------------------------------
-- 3D - Curves (given by formulas)
------------------------------------------------------------------------
makeObject(psc:PSC,tBind:BIND,1:L DROP) ==
-- obtain dependent variable and coordinate functions
t := variable tBind; tSeg := segment tBind
f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
-- create title if necessary
if not option?(l,"title" :: Symbol) then
s:String := unparse(convert(f)@InputForm)
if sayLength(s)$DisplayPackage > 50 then
  l := concat(title "AXIOM3D",l)
else l := concat(title s,l)
-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
  l := concat(style unparse(convert(f)@InputForm),l)
-- create curve with functions as coordinates
curve : PSCF := curve(makeFloatFunction(f,t),_
  makeFloatFunction(g,t),_
  makeFloatFunction(h,t))
-- call 'draw'
makeObject(curve,tSeg,1)

makeObject(psc:PSC,tBind:BIND) ==
makeObject(psc,tBind,nil())

draw(psc:PSC,tBind:BIND,l:L DROP) ==
-- obtain dependent variable and coordinate functions
t := variable tBind; tSeg := segment tBind
f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
-- create title if necessary
if not option?(l,"title" :: Symbol) then
s:String := unparse(convert(f)@InputForm)
if sayLength(s)@$DisplayPackage > 50 then
  l := concat(title "AXIOM3D",l)
else l := concat(title s,l)
-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
  l := concat(style unparse(convert(f)@InputForm),l)
-- create curve with functions as coordinates
curve : PSCF := curve(makeFloatFunction(f,t),_
  makeFloatFunction(g,t),_
  makeFloatFunction(h,t))
-- call 'draw'
draw(curve,tSeg,l)

draw(psc:PSC,tBind:BIND) ==
draw(psc,tBind,nil())

------------------------------------------------------------------------
-- 3D - Surfaces (given by formulas)
------------------------------------------------------------------------
--% Three Dimensional Function Plots
makeObject(f:Ex,xBind:BIND,yBind:BIND,l:L DROP) ==
-- create title if necessary
if not option?(l,"title" :: Symbol) then
s:String := unparse(convert(f)@InputForm)
if sayLength(s)@$DisplayPackage > 50 then
  l := concat(title "AXIOM3D",l)
else l := concat(title s,l)
-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
  l := concat(style unparse(convert(f)@InputForm),l)
-- obtain dependent variables and their ranges
x := variable xBind; xSeg := segment xBind
y := variable yBind; ySeg := segment yBind
-- call 'draw'
makeObject(makeFloatFunction(f,x,y),xSeg,ySeg,l)
makeObject(f:Ex,xBind:BIND,yBind:BIND) ==
makeObject(f,xBind,yBind,nil())

draw(f:Ex,xBind:BIND,yBind:BIND,l:L DROP) ==
  -- create title if necessary
  if not option?(l,"title" :: Symbol) then
    s:String := unparse(convert(f)@InputForm)
    if sayLength(s)$DisplayPackage > 50 then
      l := concat(title "AXIOM3D",l)
    else l := concat(title s,l)
  -- indicate draw style if necessary
  if not option?(l,"style" :: Symbol) then
    l := concat(style unparse(convert(f)@InputForm),l)
  -- obtain dependent variables and their ranges
  x := variable xBind; xSeg := segment xBind
  y := variable yBind; ySeg := segment yBind
  -- call 'draw'
draw(makeFloatFunction(f,x,y),xSeg,ySeg,l)

draw(f:Ex,xBind:BIND,yBind:BIND) ==
draw(f/xBind,yBind:BIND)

--% parametric surface

makeObject(s:PSF,uBind:BIND,vBind:BIND,l:L DROP) ==
  f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)
  if not option?(l,"title" :: Symbol) then
    s:String := unparse(convert(f)@InputForm)
    if sayLength(s)$DisplayPackage > 50 then
      l := concat(title "AXIOM3D",l)
    else l := concat(title s,l)
  if not option?(l,"style" :: Symbol) then
    l := concat(style unparse(convert(f)@InputForm),l)
  u := variable uBind; uSeg := segment uBind
  v := variable vBind; vSeg := segment vBind
  surf : PSFF := surface(makeFloatFunction(f,u,v),
                               makeFloatFunction(g,u,v),
                               makeFloatFunction(h,u,v))
  makeObject(surf,uSeg,vSeg,l)

makeObject(s:PSF,uBind:BIND,vBind:BIND) ==
makeObject(s,uBind,vBind,nil())

draw(s:PSF,uBind:BIND,vBind:BIND,l:L DROP) ==
  f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)
  -- create title if necessary
  if not option?(l,"title" :: Symbol) then
    s:String := unparse(convert(f)@InputForm)
    if sayLength(s)$DisplayPackage > 50 then
      l := concat(title "AXIOM3D",l)
else l := concat(title s,l)
-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
  l := concat(style unparse(convert(f)@InputForm),l)
-- obtain dependent variables and their ranges
u := variable uBind; uSeg := segment uBind
v := variable vBind; vSeg := segment vBind
-- create surface with functions as coordinates
surf : PSFF := surface(makeFloatFunction(f,u,v),_
    makeFloatFunction(g,u,v),_
    makeFloatFunction(h,u,v))
-- call 'draw'
draw(surf,uSeg,vSeg,l)
draw(s:PSF,uBind:BIND,vBind:BIND) ==
draw(s,uBind,vBind,nil())

——

— DRAW.dotabb —

"DRAW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAW"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DRAW" -> "ALIST"

——

package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves

—— TopLevelDrawFunctionsForAlgebraicCurves.input ——

)set break resume
)sys rm -f TopLevelDrawFunctionsForAlgebraicCurves.output
)spool TopLevelDrawFunctionsForAlgebraicCurves.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TopLevelDrawFunctionsForAlgebraicCurves
--E 1
TopLevelDrawFunctionsForAlgebraicCurves examples

TopLevelDrawFunctionsForAlgebraicCurves provides top level functions for drawing non-singular algebraic curves.

See Also:
o )show TopLevelDrawFunctionsForAlgebraicCurves

Exports:
draw

— package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves —

)abbrev package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves
++ Author: Clifton J. Williamson, Jon Steinbach
++ Date Created: 26 June 1990
++ Date Last Updated: October 1991
++ Description:
TopLevelDrawFunctionsForAlgebraicCurves(R, Ex): Exports == Implementation where
R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
Ex : FunctionSpace(R)

any1 => AnyFunctions1
drop => DrawOption
eq => Equation
f => Float
frac => Fraction
i => Integer
l => List
p => Polynomial
rn => Fraction Integer
seg => Segment
sy => Symbol
view2 => TwoDimensionalViewport

Exports => with
draw: (eq Ex, sy, sy, l drop) -> view2
++ draw(f(x,y) = g(x,y),x,y,l) draws the graph of a polynomial
++ equation. The list l of draw options must specify a region
++ in the plane in which the curve is to sketched.

Implementation => add
import ViewportPackage
import PlaneAlgebraicCurvePlot
import ViewDefaultsPackage
import GraphicsDefaults
import DrawOptionFunctions0
import SegmentFunctions2(RN,F)
import SegmentFunctions2(F,RN)
import AnyFunctions1(L SEG RN)

drawToScaleRanges: (seg f, seg f) -> l seg f
drawToScaleRanges(xVals, yVals) ==
-- warning: assumes window is square
xhi := hi xVals; xlo := lo xVals
yhi := hi yVals; ylo := lo yVals
xdiff := xhi - xlo; ydiff := yhi - ylo
pad := abs(ydiff - xdiff)/2
ydiff > xdiff =>
[segment(xlo - pad, xhi + pad), yVals]
[xVals, segment(ylo - pad, yhi + pad)]

intConvert: R -> I
intConvert r ==
\begin{verbatim}
PACKAGE DRAWCURV TOLEVELDRAWFUNCTIONSFORALGEBRAICCURVES

(nn := retractIfCan(r)@Union(I,"failed")) case "failed" =>
   error "draw: polynomial must have rational coefficients"
nn :: I

polyEquation: EQ Ex -> P I
polyEquation eq ==
   ff := lhs(eq) - rhs(eq)
   (r := retractIfCan(ff)@Union(FRAC P R,"failed")) case "failed" =>
      error "draw: not a polynomial equation"
   rat := r :: FRAC P R
   retractIfCan(denom rat)@Union(R,"failed") case "failed" =>
      error "draw: non-constant denominator"
   map(intConvert,numer rat)$PolynomialFunctions2(R,I)

draw(eq,x,y,l) ==
   -- obtain polynomial equation
   p := polyEquation eq
   -- extract ranges from option list
   floatRange := option(l,"rangeFloat" :: Symbol)
   ratRange := option(l,"rangeRat" :: Symbol)
   (floatRange case "failed") and (ratRange case "failed") =>
      error "draw: you must specify ranges for an implicit plot"
   ranges : L SEG RN := nil() -- dummy value
   floatRanges : L SEG F := nil() -- dummy value
   xRange : SEG RN := segment(0,0) -- dummy value
   yRange : SEG RN := segment(0,0) -- dummy value
   xRangeFloat : SEG F := segment(0,0) -- dummy value
   yRangeFloat : SEG F := segment(0,0) -- dummy value
   if not ratRange case "failed" then
      ranges := retract(ratRange :: Any)$ANY1(L SEG RN)
      not size?(ranges,2) => error "draw: you must specify two ranges"
      xRange := first ranges; yRange := second ranges
      xRangeFloat := map((s:RN):F+->convert(s)@Float,xRange)@(SEG F)
      yRangeFloat := map((s:RN):F+->convert(s)@Float,yRange)@(SEG F)
      floatRanges := [xRangeFloat,yRangeFloat]
   else
      floatRanges := retract(floatRange :: Any)$ANY1(L SEG F)
      not size?(floatRanges,2) =>
         error "draw: you must specify two ranges"
      xRangeFloat := first floatRanges
      yRangeFloat := second floatRanges
      xRange := map((s:F):RN+->retract(s)@RN,xRangeFloat)@(SEG RN)
      yRange := map((s:F):RN+->retract(s)@RN,yRangeFloat)@(SEG RN)
      ranges := [xRange,yRange]
   -- create curve plot
   acplot := makeSketch(p,x,y,xRange,yRange)
   -- process scaling information
   if toScale(l,drawToScale()) then
      scaledRanges := drawToScaleRanges(xRangeFloat,yRangeFloat)
      -- add scaled ranges to list of options
\end{verbatim}
```
1 := concat(ranges scaledRanges,1)
else
   -- add ranges to list of options
   l := concat(ranges floatRanges,l)
-- process color information
ptCol := pointColorPalette(l,pointColorDefault())
crCol := curveColorPalette(l,lineColorDefault())
-- draw
drawCurves(listBranches acplot,ptCol,crCol,pointSizeDefault(),1)

— DRAWCURV.dotabb —

"DRAWCURV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWCURV"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"DRAWCURV" -> "FS"

package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions

— TopLevelDrawFunctionsForCompiledFunctions.input —

)set break resume
)sys rm -f TopLevelDrawFunctionsForCompiledFunctions.output
)spool TopLevelDrawFunctionsForCompiledFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TopLevelDrawFunctionsForCompiledFunctions
--E 1

)spool
)lisp (bye)

— TopLevelDrawFunctionsForCompiledFunctions.help —
```
TopLevelDrawFunctionsForCompiledFunctions (DRAWCFUN)

Exports:
  draw  makeObject  recolor

--- package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions ---

)abbrev package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions
++ Author: Clifton J. Williamson, Scott Morrison
++ Date Created: 22 June 1990
++ Date Last Updated: January 1992
++ Description:
++ TopLevelDrawFunctionsForCompiledFunctions provides top level
++ functions for drawing graphics of expressions.

TopLevelDrawFunctionsForCompiledFunctions():
Exports == Implementation where
  ANY1 ==> AnyFunctions1
  B   ==> Boolean
  F   ==> Float
CHAPTER 21. CHAPTER T

L ==> List
SEG ==> Segment Float
SF ==> DoubleFloat
DROP ==> DrawOption
PLOT ==> Plot
PPC ==> ParametricPlaneCurve(SF -> SF)
PSC ==> ParametricSpaceCurve(SF -> SF)
PSF ==> ParametricSurface((SF,SF) -> SF)
Pt ==> Point SF
PSFUN ==> (SF, SF) -> Pt
PCFUN ==> SF -> Pt
SPACE3 ==> ThreeSpace(SF)
VIEW2 ==> TwoDimensionalViewport
VIEW3 ==> ThreeDimensionalViewport

Exports ==> with

--% Two Dimensional Function Plots

draw: (SF -> SF,SEG,L DROP) -> VIEW2
++ draw(f,a..b,l) draws the graph of \spad{y = f(x)} as x
++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.
++ The options contained in the list l of
++ the domain \spad{DrawOption} are applied.
draw: (SF -> SF,SEG) -> VIEW2
++ draw(f,a..b) draws the graph of \spad{y = f(x)} as x
++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.

--% Parametric Plane Curves

draw: (PPC,SEG,L DROP) -> VIEW2
++ draw(curve(f,g),a..b,l) draws the graph of the parametric
++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
++ \spad{max(a,b)}.
++ The options contained in the list l of the domain \spad{DrawOption}
++ are applied.
draw: (PPC,SEG) -> VIEW2
++ draw(curve(f,g),a..b) draws the graph of the parametric
++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
++ \spad{max(a,b)}.

--% Parametric Space Curves

draw: (PSC,SEG,L DROP) -> VIEW3
++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
++ curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
++ \spad{min(a,b)} to \spad{max(a,b)}.
++ The options contained in the list l of the domain
++ \spad{DrawOption} are applied.
draw: (PSC,SEG) -> VIEW3
++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
++ curve \(x = f(t), y = g(t), z = h(t)\) as t ranges from
++ \(\text{min}(a,b)\) to \(\text{max}(a,b)\).

draw: (PCFUN,SEG,L DROP) -> VIEW3
++ draw(f,a..b,l) draws the graph of the parametric
++ curve \(f\) as t ranges from
++ \(\text{min}(a,b)\) to \(\text{max}(a,b)\).
++ The options contained in the list l of the domain
++ \(\text{DrawOption}\) are applied.
draw: (PCFUN,SEG) -> VIEW3
++ draw(f,a..b,l) draws the graph of the parametric
++ curve \(f\) as t ranges from
++ \(\text{min}(a,b)\) to \(\text{max}(a,b)\).

makeObject: (PSC,SEG,L DROP) -> SPACE3
++ makeObject(curve(f,g,h),a..b,l) returns a space of the
++ domain \(\text{spadtype}(\text{ThreeSpace})\) which contains the graph of the
++ parametric curve \(\text{spad}(x = f(t), y = g(t), z = h(t))\) as t ranges from
++ \(\text{min}(a,b)\) to \(\text{max}(a,b)\);
++ The options contained in the list l of the domain
++ \(\text{DrawOption}\) are applied.

makeObject: (PSC,SEG) -> SPACE3
++ makeObject(sp,curve(f,g,h),a..b) returns the space \(\text{spad}(sp)\)
++ of the domain \(\text{spadtype}(\text{ThreeSpace})\) with the addition of the graph
++ of the parametric curve \(\text{spad}(x = f(t), y = g(t), z = h(t))\) as t
++ ranges from \(\text{spad}(\text{min}(a,b))\) to \(\text{spad}(\text{max}(a,b))\).

makeObject: (PCFUN,SEG,L DROP) -> SPACE3
++ makeObject(curve(f,g,h),a..b,l) returns a space of the
++ domain \(\text{spadtype}(\text{ThreeSpace})\) which contains the graph of the
++ parametric curve \(\text{spad}(x = f(t), y = g(t), z = h(t))\) as t ranges from
++ \(\text{min}(a,b)\) to \(\text{max}(a,b)\).
++ The options contained in the list l of the domain
++ \(\text{DrawOption}\) are applied.

makeObject: (PCFUN,SEG) -> SPACE3
++ makeObject(sp,curve(f,g,h),a..b) returns the space \(\text{spad}(sp)\)
++ of the domain \(\text{spadtype}(\text{ThreeSpace})\) with the addition of the graph
++ of the parametric curve \(\text{spad}(x = f(t), y = g(t), z = h(t))\) as t
++ ranges from \(\text{spad}(\text{min}(a,b))\) to \(\text{spad}(\text{max}(a,b))\).

--% Three Dimensional Function Plots

draw: ((SF,SF) -> SF,SEG,SEG,L DROP) -> VIEW3
++ draw(f,a..b,c..d,l) draws the graph of \(\text{spad}(z = f(x,y))\)
++ as x ranges from \(\text{spad}(\text{min}(a,b))\) to \(\text{spad}(\text{max}(a,b))\) and y ranges from
++ \(\text{spad}(\text{min}(c,d))\) to \(\text{spad}(\text{max}(c,d))\).
++ and the options contained in the list l of the domain
++ \(\text{DrawOption}\) are applied.
draw: ((SF,SF) -> SF,SEG,SEG) -> VIEW3
++ draw(f,a..b,c..d) draws the graph of \(\text{spad}(z = f(x,y))\)
++ as x ranges from \(\text{spad}(\text{min}(a,b))\) to \(\text{spad}(\text{max}(a,b))\) and y ranges from

**\spad{\texttt{makeObject}}**\((\texttt{SF,SF} \to \texttt{SF,SEG,SEG,L DROP}) \to \texttt{SPACE3}

-\% Parametric Surfaces

\texttt{draw}: \((\texttt{PSFUN}, \texttt{SEG,SEG,L DROP}) \to \texttt{VIEW3}\)

-\% Parametric Surfaces
draw: (PSF,SEG,SEG) -> VIEW3
++ draw(surface(f,g,h),a..b,c..d) draws the
++ graph of the parametric surface \(x = f(u,v), \quad y = g(u,v),\)
++ \(z = h(u,v)\) as \(u\) ranges from \(\text{min}(a,b)\) to \(\text{max}(a,b)\)
++ and \(v\) ranges from \(\text{min}(c,d)\) to \(\text{max}(c,d)\).
makeObject: (PSF,SEG,SEG,L DROP) -> SPACE3
++ makeObject(surface(f,g,h),a..b,c..d,l) returns a
++ space of the domain \(\text{spadtype}{\text{ThreeSpace}}\) which contains the
++ graph of the parametric surface \(x = f(u,v), \quad y = g(u,v),\)
++ \(z = h(u,v)\) as \(u\) ranges from \(\text{min}(a,b)\) to \(\text{max}(a,b)\)
++ \(\text{spad}(a,b)\) and \(v\) ranges from \(\text{min}(c,d)\) to \(\text{max}(c,d)\).
++ The options contained in the
++ list 1 of the domain \(\text{spad}{\text{DrawOption}}\) are applied.
makeObject: (PSF,SEG,SEG) -> SPACE3
++ makeObject(surface(f,g,h),a..b,c..d,l) returns a
++ space of the domain \(\text{spadtype}{\text{ThreeSpace}}\) which contains the
++ graph of the parametric surface \(x = f(u,v), \quad y = g(u,v),\)
++ \(z = h(u,v)\) as \(u\) ranges from \(\text{min}(a,b)\) to \(\text{max}(a,b)\)
++ \(\text{spad}(a,b)\) and \(v\) ranges from \(\text{min}(c,d)\) to \(\text{max}(c,d)\).
recolor: ((SF,SF) -> Pt,(SF,SF,SF) -> SF) -> ((SF,SF) -> Pt)
++ recolor(), uninteresting to top level user; exported in order to
++ compile package.

Implementation ==> add

I have had to work my way around the following bug in the compiler: When a local variable
is given a mapping as a value, e.g.

\[
\text{foo : SF} \to \text{SF} := \text{makeFloatFunction}(f,t),
\]

the compiler cannot distinguish that local variable from a local function defined elsewhere
in the package. Thus, when 'foo' is passed to a function, e.g.

\[
\text{bird} := \text{fcn}(\text{foo}),
\]

foo will often be compiled as DRAW;foo rather than foo. This, of course, causes a run-time
error.

To avoid this problem, local variables are not given mappings as values, but rather (single-
ton) lists of mappings. The first element of the list can always be extracted and everything
goes through as before. There is no major loss in efficiency, as the computation of points
will always dominate the computation time.

- cjw, 22 June MCMXC

— package DRAWCfun TopLevelDrawFunctionsForCompiledFunctions —

import PLOT
import TwoDimensionalPlotClipping
import GraphicsDefaults
import ViewportPackage
import ThreeDimensionalViewport
import DrawOptionFunctions0
import MakeFloatCompiledFunction(Ex)
import MeshCreationRoutinesForThreeDimensions
import SegmentFunctions2(SF,Float)
import ViewDefaultsPackage
import AnyFunctions1(Pt -> Pt)
import AnyFunctions1((SF,SF,SF) -> SF)
import DrawOptionFunctions0
import SPACE3

EXTOVARERROR : String := 
"draw: when specifying function, left hand side must be a variable"
SMALLRANGEERROR : String := 
"draw: range is in interval with only one point"
DEPVARERROR : String := 
"draw: independent variable appears on lhs of function definition"

------------------------------------------------------------------------
-- 2D - draw's
------------------------------------------------------------------------

drawToScaleRanges: (Segment SF,Segment SF) -> L SEG
drawToScaleRanges(xVals,yVals) ==
  -- warning: assumes window is square
  xHi := convert(hi xVals)@Float; xLo := convert(lo xVals)@Float
  yHi := convert(hi yVals)@Float; yLo := convert(lo yVals)@Float
  xDiff := xHi - xLo; yDiff := yHi - yLo
  pad := abs(yDiff - xDiff)/2
  yDiff > xDiff =>
    [segment(xLo - pad,xHi + pad),map(x +-> convert(x)@Float,yVals)]
  [map(x +-> convert(x)@Float,xVals),segment(yLo - pad,yHi + pad)]

drawPlot: (PLOT,L DROP) -> VIEW2
drawPlot(plot,l) ==
  branches := listBranches plot
  xRange := xRange plot; yRange := yRange plot
  -- process clipping information
  if (c1 := option(l,"clipSegment" :: Symbol)) case "failed" then
    if clipBoolean(1,clipPointsDefault()) then
      clipInfo :=
        parametric? plot => clipParametric plot
        clip plot
        branches := clipInfo.branst
        xRange := clipInfo.xValues; yRange := clipInfo.yValues
    else
      "No explicit user-specified clipping"
  else
    ...
segList := retract(cl :: Any)$ANY1(L SEG)
empty? segList =>
   error "draw: you may specify at least 1 segment for 2D clipping"
more?(segList,2) =>
   error "draw: you may specify at most 2 segments for 2D clipping"
xLo : SF := 0; xHi : SF := 0; yLo : SF := 0; yHi : SF := 0
if empty? rest segList then
   xLo := lo xRange; xHi := hi xRange
   yRangeF := first segList
   yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
else
   xRangeF := first segList
   xLo := convert(lo xRangeF)@SF; xHi := convert(hi xRangeF)@SF
   yRangeF := second segList
   yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
   clipInfo := clipWithRanges(branches,xLo,xHi,yLo,yHi)
   branches := clipInfo.brans
   xRange := clipInfo.xValues; yRange := clipInfo.yValues
-- process scaling information
if toScale(l,drawToScale()) then
   scaledRanges := drawToScaleRanges(xRange,yRange)
   -- add scaled ranges to list of options
   l := concat(ranges scaledRanges,l)
else
   xRangeFloat : SEG := map(x +-> convert(x)@Float,xRange)
   yRangeFloat : SEG := map(x +-> convert(x)@Float,yRange)
   -- add ranges to list of options
   l := concat(ranges(ll : L SEG := [xRangeFloat,yRangeFloat]),l)
-- process color information
ptCol := pointColorPalette(l,pointColorDefault())
crCol := curveColorPalette(l,lineColorDefault())
-- draw
drawCurves(branches,ptCol,crCol,pointSizeDefault(),l)

normalize: SEG -> Segment SF
normalize seg ==
   -- normalize [a,b]:
   -- error if a = b, returns [a,b] if a < b, returns [b,a] if b > a
   a := convert(lo seg)@SF; b := convert(hi seg)@SF
   a = b => error SMALLRANGEERROR
   a < b => segment(a,b)
   segment(b,a)

|||-
The function myTrap1 is a local function for used in creating maps SF \rightarrow Point SF (two dimensional). The range of this function is SingleFloat. As originally coded it would return \$NaN\$ value Lisp which is outside the range. Since this function is only used internally by the draw package we handle the “failed” case by returning zero. We handle the out-of-range
case by returning the maximum or minimum SingleFloat value.

— package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions —

```lisp
myTrap1: (SF-> SF, SF) -> SF
myTrap1(ff:SF-> SF, f:SF):SF ==
  s := trapNumericErrors(ff(f))$Lisp :: Union(SF, "failed")
  s case "failed" => 0
  r := s::SF
  if r > max()$SF => max()$SF
  if r < min()$SF => min()$SF
  r

makePt2: (SF,SF) -> Point SF
makePt2(x,y) == point(l : List SF := [x,y])
```

```lisp
--% Two Dimensional Function Plots

draw(f:SF -> SF,seg:SEG,l:L DROP) ==
  -- set adaptive plotting off or on
  oldAdaptive := adaptive?()$PLOT
  setAdaptive(adaptive(l,oldAdaptive))$PLOT
  -- create function SF -> Point SF
  ff : L(SF -> Point SF) := [x +-> makePt2(myTrap1(f,x),x)]
  -- process change of coordinates
  if (c := option(l,"coordinates" :: Symbol)) case "failed" then
    -- default coordinate transformation
    ff := [x +-> makePt2(x,myTrap1(f,x))]
  else
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    ff := [x +-> (first cc)((first ff)(x))]
  -- create PLG
  pl := pointPlot(first ff,normalize seg)
  -- reset adaptive plotting
  setAdaptive(oldAdaptive)$PLOT
  -- draw
  drawPlot(pl,l)

draw(f:SF -> SF,seg:SEG) == draw(f,seg,nil())
```

```lisp
--% Parametric Plane Curves

draw(ppc:PPC,seg:SEG,l:L DROP) ==
  -- set adaptive plotting off or on
  oldAdaptive := adaptive?()$PLOT
  setAdaptive(adaptive(l,oldAdaptive))$PLOT
  -- create function SF -> Point SF
  f := coordinate(ppc,1); g := coordinate(ppc,2)
  fcn : L(SF -> Pt) := [x +-> makePt2(myTrap1(f,x),myTrap1(g,x))]
  -- process change of coordinates
```

```lisp
```
if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
c := L(Pt -> Pt) := [retract(c :: Any)@ANY1(Pt -> Pt)]
fcn := [x +-> (first cc)((first fcn)(x))]
-- create PLOT
pl := pointPlot(first fcn,normalize seg)
-- reset adaptive plotting
setAdaptive(oldAdaptive)$PLOT
-- draw
drawPlot(pl,l)

draw(ppc:PPC,seg:SEG) == draw(ppc,seg,nil())

-- 3D - Curves

--% functions for creation of maps SF -> Point SF (three dimensional)

makePt4: (SF,SF,SF,SF) -> Point SF
makePt4(x,y,z,c) == point(l : List SF := [x,y,z,c])

--% Parametric Space Curves

id: SF -> SF
id x == x

zCoord: (SF,SF,SF) -> SF
zCoord(x,y,z) == z

colorPoints: (List List Pt,(SF,SF,SF) -> SF) -> List List Pt
colorPoints(llp,func) ==
  for lp in llp repeat for p in lp repeat
    p.4 := func(p.1,p.2,p.3)
  llp

makeObject(psc:PSC,seg:SEG,l:L DROP) ==
  sp := space l
  -- obtain dependent variable and coordinate functions
  f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
  -- create function SF -> Point SF with default or user-specified
  -- color function
  fcn : L(SF -> Pt) := [x +-> makePt4(myTrap1(f,x),myTrap1(g,x),
                       myTrap1(h,x), myTrap1(id,x))]
  pointsColored? : Boolean := false
  if not (c1 := option(l,"colorFunction1" :: Symbol)) case "failed" then
    pointsColored? := true
    fcn := [x +-> makePt4(myTrap1(f,x),myTrap1(g,x),myTrap1(h,x),
                      retract(c1 :: Any)@ANY1(SF -> SF)(x))]
  -- process change of coordinates
  if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
CHAPTER 21. CHAPTER T

```lisp
cc : L(Pt -> Pt) := \(\text{retract}(c :: \text{Any})@\text{ANY1}(\text{Pt} \to \text{Pt})\)
fcn := \(x \mapsto (\text{first cc})(\text{first fcn}(x))\)

-- create PLOT
pl := \text{pointPlot}(\text{first fcn}, \text{normalize seg})@\text{Plot3D}

-- create ThreeSpace
s := sp

-- draw Tube

\text{option?(l, "tubeRadius" :: Symbol) =>}
pts := \text{tubePoints}(1, 8)
rad := \text{convert}(\text{tubeRadius}(1, 0.25))@\text{DoubleFloat}
tub := \text{tub}(pl, rad, pts)@\text{NumericTubePlot}(\text{Plot3D})
loops := \text{listLoops} tub

-- color points if this has not been done already
if not pointsColored? then
  if (\(c3 := \text{option}(l, "colorFunction3" :: Symbol)\)) case "failed"
    then \text{colorPoints(loops, zCoord)} -- default color function
    else \text{colorPoints(loops, retract(c3 :: Any)@\text{ANY1}((\text{SF}, \text{SF}, \text{SF}) \to \text{SF})\)}

mesh(s, loops, false, false)

s

-- draw curve
br := \text{listBranches} pl
for b in br repeat \text{curve}(s, b)

s

\text{makeObject}(psc: \text{PCFUN}, \text{seg: SEG}, 1: L \text{ DROP}) ==

sp := \text{space} l

-- create function SF -> Point SF with default or user-specified
-- color function
fcn := \(L(\text{SF} \to \text{Pt}) := [psc]\)

pointsColored? : Boolean := false
if not (\(c1 := \text{option}(l, "colorFunction1" :: Symbol)\)) case "failed" then
  pointsColored? := true
fcn := \(x \mapsto \text{concat}(psc(x), \text{retract}(c1 :: \text{Any})@\text{ANY1}((\text{SF} \to \text{SF})(x))\))

-- process change of coordinates
if not (\(c := \text{option}(l, "coordinates" :: Symbol)\)) case "failed" then
  cc := \(L(\text{Pt} \to \text{Pt}) := [\text{retract}(c :: \text{Any})@\text{ANY1}(\text{Pt} \to \text{Pt})]\)
fcn := \(x \mapsto (\text{first cc})(\text{first fcn}(x))\)

-- create PLOT
pl := \text{pointPlot}(\text{first fcn}, \text{normalize seg})@\text{Plot3D}

-- create ThreeSpace
s := sp

-- draw Tube

\text{option?(l, "tubeRadius" :: Symbol) =>}
pts := \text{tubePoints}(1, 8)
rad := \text{convert}(\text{tubeRadius}(1, 0.25))@\text{DoubleFloat}
tub := \text{tub}(pl, rad, pts)@\text{NumericTubePlot}(\text{Plot3D})
loops := \text{listLoops} tub

-- color points if this has not been done already
mesh(s, loops, false, false)
```
s
-- draw curve
br := listBranches pl
for b in br repeat curve(s,b)
s
makeObject(psc:PSC,seg:SEG) ==
makeObject(psc,seg,nil())

makeObject(psc:PCFUN,seg:SEG) ==
makeObject(psc,seg,nil())

draw(psc:PSC,seg:SEG,1:L DROP) ==
sp := makeObject(psc,seg,1)
makeViewport3D(sp, 1)
draw(psc:PSC,seg:SEG) ==
draw(psc,seg,nil())
draw(psc:PCFUN,seg:SEG,1:L DROP) ==
sp := makeObject(psc,seg,1)
makeViewport3D(sp, 1)
draw(psc:PCFUN,seg:SEG) ==
draw(psc,seg,nil())

------------------------------------------------------------------------
-- 3D - Surfaces
------------------------------------------------------------------------

The function myTrap2 is a local function for used in creating maps SF \rightarrow Point SF (three dimensional). The range of this function is SingleFloat. As originally coded it would return $\text{NaN value}$ Lisp which is outside the range. Since this function is only used internally by the draw package we handle the “failed” case by returning zero. We handle the out-of-range case by returning the maximum or minimum SingleFloat value.

— package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions —

myTrap2: ((SF, SF) \rightarrow SF, SF, SF) \rightarrow SF
myTrap2(ff:(SF, SF) \rightarrow SF, u:SF, v:SF):SF ==
s := trapNumericErrors(ff(u, v))$Lisp :: Union(SF, "failed")
s case "failed" => 0
r:SF := s::SF
r > max()$SF => max()$SF
r < min()$SF => min()$SF
r
recolor(ptFunc, colFunc) ==
  (f1, f2) ->
  pt := ptFunc(f1, f2)
  pt.4 := colFunc(pt.1, pt.2, pt.3)
  pt

xCoord: (SF, SF) -> SF
xCoord(x, y) == x

--% Three Dimensional Function Plots

makeObject(f: (SF, SF) -> SF, xSeg: SEG, ySeg: SEG, l:L DROP) ==
  sp := space l
  -- process color function of two variables
  col2 : L((SF, SF) -> SF) := [xCoord]  -- dummy color function
  pointsColored?: Boolean := false
  if not (c2 := option(l, "colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF, SF) -> SF)]
  fcn : L((SF, SF) -> Pt) :=
  [(x, y) +-> makePt4(myTrap2(f, x, y), x, y, (first col2)(x, y))]
  -- process change of coordinates
  if (c := option(l, "coordinates" :: Symbol)) case "failed" then
    -- default coordinate transformation
    fcn := [(x, y) +-> makePt4(x, y, myTrap2(f, x, y), (first col2)(x, y))]
  else
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [(x, y) +-> (first cc)((first fcn)(x, y))]
  -- process color function of three variables, if there was no
  -- color function of two variables
  if not pointsColored? then
    c := option(l, "colorFunction3" :: Symbol)
    fcn :=
    c case "failed" => [recolor((first fcn), zCoord)]
    [recolor((first fcn), retract(c :: Any)$ANY1((SF, SF, SF) -> SF))]
  -- create mesh
  mesh := meshPar2Var(sp, first fcn, normalize xSeg, normalize ySeg, l)
  mesh

makeObject(f: (SF, SF) -> SF, xSeg: SEG, ySeg: SEG) ==
  makeObject(f, xSeg, ySeg, nil())

draw(f: (SF, SF) -> SF, xSeg: SEG, ySeg: SEG, l:L DROP) ==
  sp := makeObject(f, xSeg, ySeg, l)
  makeViewport3D(sp, l)

draw(f: (SF, SF) -> SF, xSeg: SEG, ySeg: SEG) ==
  draw(f, xSeg, ySeg, nil())

--% parametric surface
makeObject(s:PSF,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  sp := space l
  -- create functions from expressions
  f : L((SF,SF) -> SF) := [coordinate(s,1)]
  g : L((SF,SF) -> SF) := [coordinate(s,2)]
  h : L((SF,SF) -> SF) := [coordinate(s,3)]
  -- process color function of two variables
  col2 : L((SF,SF) -> SF) := [xCoord]  -- dummy color function
  pointsColored? : Boolean := false  -- dummy color function
  if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
  fcn : L((SF,SF) -> Pt) :=
    [(x,y) +-> makePt4(myTrap2((first f),x,y),myTrap2((first g),x,y),
      myTrap2((first h),x,y), myTrap2((first col2),x,y))]
  -- process change of coordinates
  if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
  -- process color function of three variables, if there was no
  -- color function of two variables
  if not pointsColored? then
    col3 : L((SF,SF,SF) -> SF) := [zCoord]  -- default color function
    if not (c := option(l,"colorFunction3" :: Symbol)) case "failed" then
      col3 := [retract(c :: Any)$ANY1((SF,SF,SF) -> SF)]
      fcn := [recolor((first fcn),(first col3))]
  -- create mesh
  mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
  mesh

makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  sp := space l
  -- process color function of two variables
  col2 : L((SF,SF) -> SF) := [xCoord]  -- dummy color function
  pointsColored? : Boolean := false  -- dummy color function
  if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
  fcn : L((SF,SF) -> Pt) :=
    pointsColored? => [(x,y) +-> concat(s(x, y), (first col2)(x, y))]
  -- process change of coordinates
  if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
  -- create mesh
  mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
  mesh
makeObject(s:PSF,uSeg:SEG,vSeg:SEG) ==
    makeObject(s,uSeg,vSeg,nil())

draw(s:PSF,uSeg:SEG,vSeg:SEG,1:L DROP) ==
    -- draw
    mesh := makeObject(s,uSeg,vSeg,1)
    makeViewport3D(mesh,1)

draw(s:PSF,uSeg:SEG,vSeg:SEG) ==
    draw(s,uSeg,vSeg,nil())

makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
    makeObject(s,uSeg,vSeg,nil())

draw(s:PSFUN,uSeg:SEG,vSeg:SEG,1:L DROP) ==
    -- draw
    mesh := makeObject(s,uSeg,vSeg,1)
    makeViewport3D(mesh,1)

draw(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
    draw(s,uSeg,vSeg,nil())

package DRAWPT TopLevelDrawFunctionsForPoints

)set break resume
)sys rm -f TopLevelDrawFunctionsForPoints.output
)spool TopLevelDrawFunctionsForPoints.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
TopLevelDrawFunctionsForPoints (DRAWPT)

Exports:
draw

--- package DRAWPT TopLevelDrawFunctionsForPoints ---

)abbrev package DRAWPT TopLevelDrawFunctionsForPoints
++ Author: Mike Dewar
++ Date Created: 24 May 1995
TopLevelDrawFunctionsForPoints(): Exports == Implementation where

DROPOutput ==> DrawOption
L ==> List
SF ==> DoubleFloat
Pt ==> Point SF
VIEW2 ==> TwoDimensionalViewport
VIEW3 ==> ThreeDimensionalViewport

Exports == with

draw: (L SF, L SF) -> VIEW2
  ++ draw(lx,ly) plots the curve constructed of points (x,y) for x
  ++ in \spad{lx} for y in \spad{ly}.

draw: (L SF, L SF, L DROP) -> VIEW2
  ++ draw(lx,ly,l) plots the curve constructed of points (x,y) for x
  ++ in \spad{lx} for y in \spad{ly}.
  ++ The options contained in the list l of
  ++ the domain \spad{DrawOption} are applied.

draw: (L Pt) -> VIEW2
  ++ draw(lp) plots the curve constructed from the list of points lp.

draw: (L Pt, L DROP) -> VIEW2
  ++ draw(lp,l) plots the curve constructed from the list of points lp.
  ++ The options contained in the list l of the domain \spad{DrawOption}
  ++ are applied.

draw: (L SF, L SF, L SF) -> VIEW3
  ++ draw(lx,ly,lz) draws the surface constructed by projecting the values
  ++ in the \axiom{lz} list onto the rectangular grid formed by the
  ++ \axiom{lx x ly}.

draw: (L SF, L SF, L SF, L DROP) -> VIEW3
  ++ draw(lx,ly,lz,l) draws the surface constructed by
  ++ projecting the values
  ++ in the \axiom{lz} list onto the rectangular grid formed by the
  ++ The options contained in the list l of the domain \spad{DrawOption}
  ++ are applied.

Implementation == add

draw(lp:L Pt, l: L DROP): VIEW2 ==
  makeViewport2D(makeGraphImage([lp])$GraphImage, l)$VIEW2

draw(lp:L Pt): VIEW2 == draw(lp, [])

draw(lx: L SF, ly: L SF, l: L DROP): VIEW2 ==
  draw([point([x, y])$Pt for x in lx for y in ly], l)

...
PACKAGE TOPSP TOPLEVELTHREESPACE

---

DRAWPPT.dotabb

package TOPSP TOPLevelThreeSpace

---

)set break resume
)sys rm -f TopLevelThreeSpace.output
)spool TopLevelThreeSpace.output
)set message test on
TopLevelThreeSpace (TOPSP)

Exports:
createThreeSpace

--- package TOPSP TopLevelThreeSpace ---

)abbrev package TOPSP TopLevelThreeSpace
++ Description:
++ This package exports a function for making a \spadtype{ThreeSpace}

TopLevelThreeSpace(): with
  createThreeSpace: () -> ThreeSpace DoubleFloat
  ++ createThreeSpace() creates a \spadtype{ThreeSpace(DoubleFloat)} object
  ++ capable of holding point, curve, mesh components and any combination.
  == add
  createThreeSpace() == create3Space()$ThreeSpace(DoubleFloat)

——

— TOPSP.dotabb —

"TOPSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOPSP"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"TOPSP" -> "FIELD"
"TOPSP" -> "RADCAT"

——

package INTHERTR TranscendentalHermiteIntegration

—— TranscendentalHermiteIntegration.input ——

)set break resume
)sys rm -f TranscendentalHermiteIntegration.output
)spool TranscendentalHermiteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalHermiteIntegration
--E 1

)spool
)lisp (bye)

——

— TranscendentalHermiteIntegration.help ——
TranscendentalHermiteIntegration examples

Hermite integration, transcendental case.

See Also:
- show TranscendentalHermiteIntegration
Exports ==> with
   HermiteIntegrate: (RF, UP -> UP) -> HER
   ++ HermiteIntegrate(f, D) returns \spad{[g, h, s, p]}
   ++ such that \spad{f = Dg + h + s + p},
   ++ h has a squarefree denominator normal w.r.t. D,
   ++ and all the squarefree factors of the denominator of s are
   ++ special w.r.t. D. Furthermore, h and s have no polynomial parts.
   ++ D is the derivation to use on \spadtype{UP}.

Implementation ==> add
   import MonomialExtensionTools(F, UP)

   normalHermiteIntegrate: (RF,UP->UP) -> Record(answer:RF,lognum:UP,logden:UP)

   HermiteIntegrate(f, derivation) ==
      rec := decompose(f, derivation)
      hi := normalHermiteIntegrate(rec.normal, derivation)
      qr := divide(hi.lognum, hi.logden)
      [hi.answer, qr.remainder / hi.logden, rec.special, qr.quotient + rec.poly]

   normalHermiteIntegrate(f, derivation) ==
      a := numer f
      q := denom f
      p:UP := 0
      mult:UP := 1
      qhat := (q exquo (g0 := g := gcd(q, differentiate q)))::UP
      while(degree(qbar := g) > 0) repeat
         qbarhat := (qbar exquo (g := gcd(qbar, differentiate qbar)))::UP
         qtil := - ((qhat * (derivation qbar)) exquo qbar)::UP
         bc :=
            extendedEuclidean(qtil, qbarhat, a)::Record(coef1:UP, coef2:UP)
         qr := divide(bc.coef1, qbarhat)
         a := bc.coef2 + qtil * qr.quotient - derivation(qr.remainder)
            * (qhat exquo qbarhat)::UP
         p := p + mult * qr.remainder
         mult := mult * qbarhat
      [p / g0, a, qhat]
package INTTR TranscendentalIntegration

--- TranscendentalIntegration.input ---

)set break resume
)sys rm -f TranscendentalIntegration.output
)spool TranscendentalIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalIntegration
--E 1

)spool
)lisp (bye)

--- TranscendentalIntegration.help ---

====================================================================
TranscendentalIntegration examples
====================================================================

This package provides functions for the transcendental case of the
Risch algorithm.

See Also:
  • )show TranscendentalIntegration
TranscendentalIntegration (INTTR)

Exports:

- expextendedint
- expintfdpoly
- monomialIntPoly
- primextendedint
- primintegrate
- primlimintfrac
- tanintegrate
- expintegrate
- explimitedint
- monomialIntegrate
- primextintfrac
- primintfdpoly
- primlimitedint

--- package INTTR TranscendentalIntegration ---

)abbrev package INTTR TranscendentalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ This package provides functions for the transcendental
++ case of the Risch algorithm.
++ Internally used by the integrator

TranscendentalIntegration(F, UP): Exports == Implementation where
F : Field
UP : UnivariatePolynomialCategory F

N  ==> NonNegativeInteger
Z  ==> Integer
Q  ==> Fraction Z
GP ==> LaurentPolynomial(F, UP)
UP2 ==> SparseUnivariatePolynomial UP
RF ==> Fraction UP
UPR ==> SparseUnivariatePolynomial RF
IR ==> IntegrationResult RF
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
LLG ==> List Record(coeff:RF, logand:RF)
NE ==> Record(integrand:RF, intvar:RF)
NL ==> Record(mainpart:RF, limitedlogs:LLG)
UPF ==> Record(answer:UP, a0:F)
RFF ==> Record(answer:RF, a0:F)
IRF ==> Record(answer:IR, a0:F)
NLF ==> Record(answer:NL, a0:F)
GPF ==> Record(answer:GP, a0:F)
UPUP ==> Record(elem:UP, notelem:UP)
GPGP ==> Record(elem:GP, notelem:GP)
RFRF ==> Record(elem:RF, notelem:RF)
FF ==> Record(ratpart:F, coeff:F)
FRF ==> Record(ratpart:RF, coeff:RF)
UF ==> Union(FF, "failed")
UF2 ==> Union(List F, "failed")
REC ==> Record(ir:IR, specpart:RF, polypart:UP)
PSOL ==> Record(ans:F, right:F, sol?:Boolean)
FAIL ==> error "Sorry - cannot handle that integrand yet"

Exports ==> with
primintegrate : (RF, UP -> UP, F -> UF) -> IRF
++ primintegrate(f, ', foo) returns \spad{\{g, a\}} such that
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in UP.
++ Argument foo is an extended integration function on F.
expintegrate : (RF, UP -> UP, (Z, F) -> PSOL) -> IRF
++ expintegrate(f, ', foo) returns \spad{\{g, a\}} such that
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F;
++ Argument foo is a Risch differential equation solver on F;
tanintegrate : (RF, UP -> UP, (Z, F, F) -> UF) -> IRF
++ tanintegrate(f, ', foo) returns \spad{\{g, a\}} such that
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F;
++ Argument foo is a Risch differential system solver on F;
primextendedint:(RF, UP -> UP, F->UF, RF) -> Union(RFF,FFR,"failed")
++ primextendedint(f, ', foo, g) returns either \spad{\{v, c\}} such that
++ \spad{f = v' + c g} and \spad{c' = 0}, or \spad{\{v, a\}} such that
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in UP.
++ Returns "failed" if neither case can hold.
++ Argument foo is an extended integration function on F.
expextendedint:(RF,UP->UP,(Z,F)->PSOL, RF) -> Union(RFF,FFR,"failed")
++ expextendedint(f, ', foo, g) returns either \spad{\{v, c\}} such that
++ \spad{f = v' + c g} and \spad{c' = 0}, or \spad{\{v, a\}} such that
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F.
++ Returns "failed" if neither case can hold.
++ Argument foo is a Risch differential equation function on F.
primlimitedint:(RF, UP -> UP, F->UF, List RF) -> Union(NLF,"failed")
++ primlimitedint(f, ', foo, [ui,...,un]) returns
++ \spad{\{v, [c1,...,cn], a\}} such that \spad{\{ci' = 0\},
++ \spad{f = v' + a + reduce(+,[ci * ui'/ui])}},
++ and \spad{a = 0} or \spad{a} has no integral in UP.
++ Returns "failed" if no such v, ci, a exist.
++ Argument foo is an extended integration function on F.
explimitedint:(RF, UP->UP,(Z,F)->PSOL,List RF) -> Union(NLF,"failed")
++ explimitedint(f, ', foo, [u1,...,un]) returns
++ \[v, [c1,...,cn], a\] such that \spad{ci' = 0},
++ \spad{f = v' + a + reduce(*,[ci * ui'/ui])},
++ and \spad{a = 0} or \spad{a} has no integral in F.
++ Returns "failed" if no such v, ci, a exist.
++ Argument foo is a Risch differential equation function on F.
primextintfrac : (RF, UP -> UP, RF) -> Union(FFR, "failed")
++ primextintfrac(f, ', g) returns \[v, c\] such that
++ \spad{f = v' + c g} and \spad{c' = 0}.
++ Error: if \spad{degree numer f >= degree denom f} or
++ if \spad{degree numer g >= degree denom g} or
++ if \spad{denom g} is not squarefree.
primlimintfrac : (RF, UP -> UP, List RF) -> Union(NL, "failed")
++ primlimintfrac(f, ', [u1,...,un]) returns \[v, [c1,...,cn]\]
++ such that \spad{ci' = 0} and \spad{f = v' + +/[ci * ui'/ui]}.
++ Error: if \spad{degree numer f >= degree denom f}.
primintfldpoly : (UP, F -> UF, F) -> Union(UP, "failed")
++ primintfldpoly(p, ', t') returns q such that \spad{p' = q} or
++ "failed" if no such q exists. Argument \spad{t'} is the derivative of
++ the primitive generating the extension.
expintfldpoly : (GP, (Z, F) -> PSOL) -> Union(GP, "failed")
++ expintfldpoly(p, foo) returns q such that \spad{p' = q} or
++ "failed" if no such q exists.
++ Argument foo is a Risch differential equation function on F.
monomialIntegrate : (RF, UP -> UP) -> REC
++ monomialIntegrate(f, ') returns \spad{[ir, s, p]} such that
++ \spad{f = ir' + s + p} and all the squarefree factors of the
++ denominator of s are special w.r.t the derivation '.
monomialIntPoly : (UP, UP -> UP) -> Record(answer:UP, polypart:UP)
++ monomialIntPoly(p, ') returns [q, r] such that
++ \spad{p = q' + r} and \spad{degree(r) < degree(t')}.
++ Error if \spad{degree(t') < 2}.

Implementation ==> add
import SubResultantPackage(UP, UP2)
import MonomialExtensionTools(F, UP)
import TranscendentalHermiteIntegration(F, UP)
import CommuteUnivariatePolynomialCategory(F, UP, UP2)

primintegratepoly : (UP, F -> UF, F) -> Union(UPF, UPUP)
expintegratepoly : (GP, (Z, F) -> PSOL) -> Union(GPF, GPGP)
expextintfrac : (RF, UP -> UP, RF) -> Union(FFR, "failed")
explimintfrac : (RF, UP -> UP, List RF) -> Union(NL, "failed")
limitedLogs : (RF, RF -> RF, List RF) -> Union(LLG, "failed")
logprmderiv : (RF, UP -> UP) -> RF
logexpderiv : (RF, UP -> UP, F) -> RF
tanintegratespecial: (RF, RF -> RF, (Z, F, F) -> UF2) -> Union(RFF, RFRF)
UP2UP2 : UP -> UP2
UP2UPR : UP -> UPR
UP22UPR : UP2 -> UPR
\[
\text{notelementary} : \text{REC} \rightarrow \text{IR} \\
\text{kappa} : (\text{UP}, \text{UP} \rightarrow \text{UP}) \rightarrow \text{UP} \\
\text{dummy:RF} := 0 \\
\text{logprmderiv}(f, \text{derivation}) = \text{differentiate}(f, \text{derivation}) / f \\
\text{UP2UP2} p := \\
\text{map}(x \mapsto x::\text{UP}, p)$\text{UnivariatePolynomialCategoryFunctions2}(\text{F}, \text{UP}, \text{UP}, \text{UP2}) \\
\text{UP2UPR} p := \\
\text{map}(x \mapsto x::\text{UP}::\text{RF}, p)$\text{UnivariatePolynomialCategoryFunctions2}(\text{F}, \text{UP}, \text{RF}, \text{UPR}) \\
\text{UP22UPR} p := \\
\text{map}(x \mapsto x::\text{RF}, p)$\text{SparseUnivariatePolynomialFunctions2}(\text{UP}, \text{RF}) \\
\text{kappa}(p, \text{derivation}) = \\
\text{ans:UP} := 0 \\
\text{while} p \neq 0 \text{ repeat} \\
\text{ans} := \text{ans} + \text{derivation}(\text{leadingCoefficient}(p)::\text{UP}) \cdot \text{monomial}(1, \text{degree} p) \\
p := \text{reductum} p \\
\text{ans} \\
\text{monomialIntegrate}(f, \text{derivation}) = \\
\text{zero? f} => [0, 0, 0] \\
r := \text{HermiteIntegrate}(f, \text{derivation}) \\
\text{zero?}(\text{inum} := \text{numer}(r.\text{logpart})) => [r.\text{answer}::\text{IR}, r.\text{specpart}, r.\text{polypart}] \\
\text{iden} := \text{denom}(r.\text{logpart}) \\
x := \text{monomial}(1, 1)$\text{UP} \\
\text{resultvec} := \text{subresultantVector}(\text{UP2UP2} \text{inum} - \\
(x::\text{UP2}) \cdot \text{UP2UP2} \text{derivation} \text{iden}, \text{UP2UP2} \text{iden}) \\
\text{respoly} := \text{primitivePart} \text{leadingCoefficient} \text{resultvec} 0 \\
\text{rec} := \text{splitSquarefree}(\text{respoly}, x1 \mapsto \text{kappa}(x1, \text{derivation})) \\
\text{logs:List(LOG)} := [ \\
[1, \text{UP2UPR}(\text{term.factor}), \\
\text{UP2UPR} \text{swap} \text{primitivePart}(\text{resultvec}(\text{term.exponent}), \text{term.factor})] \\
\text{for term in factors}(\text{rec.special})] \\
\text{dlog} := \\
\text{one? derivation x} \Rightarrow r.\text{logpart} \\
((\text{derivation} x) = 1) \Rightarrow r.\text{logpart} \\
\text{differentiate}(\text{mkAnswer}(0, \text{logs}, \text{empty}()),) \\
(x1:RF):RF \mapsto \text{differentiate}(x1, \text{derivation})) \\
(u := \text{retractIfCan}(p := r.\text{logpart} - \text{dlog})$\text{Union}(\text{UP}, "failed")) \text{ case UP} => \\
[\text{mkAnswer}(r.\text{answer}, \text{logs}, \text{empty}), r.\text{specpart}, r.\text{polypart} + u::UP] \\
[\text{mkAnswer}(r.\text{answer}, \text{logs}, [[p, \text{dummy}]]), r.\text{specpart}, r.\text{polypart}] \\
\text{-- returns [q, r] such that p = q' + r and degree(r) < degree(dt)
-- must have degree(derivation \( t \)) \( \geq 2 \)

```lisp
monomialIntPoly(p, derivation) ==
    (d := degree(dt := derivation monomial(1,1))::Z) < 2 =>
    error "monomIntPoly: monomial must have degree 2 or more"
    l := leadingCoefficient dt
    ans:UP := 0
    while (n := 1 + degree(p)::Z - d) > 0 repeat
        ans := ans + (term := monomial(leadingCoefficient(p) / (n * l), n::N))
        p := p - derivation term -- degree(p) must drop here
    [ans, p]
```

-- returns either

-- \((q \in \text{GP}, a \in \text{F})\) s.t. \( p = q' + a \), and \( a=0 \) or \( a \) has no integral in \( \text{F} \)
-- or \((q \in \text{GP}, r \in \text{GP})\) s.t. \( p = q' + r \), and \( r \) has no integral elem/\( \text{UP} \)

```lisp
expintegratepoly(p, FRDE) ==
    coef0:F := 0
    notelm := answr := 0$GP
    while p ^= 0 repeat
        ans1 := FRDE(n := degree p, a := leadingCoefficient p)
        answr := answr + monomial(ans1.ans, n)
        if "ans1.sol?" then -- Risch d.e. has no complete solution
            missing := a - ans1.right
            if zero? n then coef0 := missing
            else notelm := notelm + monomial(missing, n)
        p := reductum p
    zero? notelm => [answr, coef0]
    [answr, notelm]
```

-- f is either 0 or of the form \( p(t)/(1 + t**2)**n \)
-- returns either

-- \((q \in \text{RF}, a \in \text{F})\) s.t. \( f = q' + a \), and \( a=0 \) or \( a \) has no integral in \( \text{F} \)
-- or \((q \in \text{RF}, r \in \text{RF})\) s.t. \( f = q' + r \), and \( r \) has no integral elem/\( \text{UP} \)

```lisp
tanintegratespecial(f, derivation, FRDE) ==
    ans:RF := 0
    p := monomial(1, 2)$UP + 1
    while (n := degree(denom f) quo 2) ^= 0 repeat
        r := numer(f) rem p
        a := coefficient(r, 1)
        b := coefficient(r, 0)
        (u := FRDE(n, a, b)) case "failed" => return [ans, f]
        l := u::List(F)
        term:RF := (monomial(first l, 1)$UP + second(l)::UP) / denom f
        ans := ans + term
        f := f - derivation term -- the order of the pole at 1+t**2 drops
    zero?(c0 := retract(retract(f)@UP)@F) or
    (u := FRDE(0, c0, 0)) case "failed" => [ans, c0]
    [ans + first(u::List(F))::UP::RF, 0::F]
```

-- returns \((v \in \text{RF}, c \in \text{RF})\) s.t. \( f = v' + cg \), and \( c' = 0 \), or "failed"
-- g must have a squarefree denominator (always possible)
--- g must have no polynomial part and no pole above \( t = 0 \)
--- f must have no polynomial part and no pole above \( t = 0 \)

\[
\text{expextintfrac}(f, \text{derivation}, g) = \\
\text{zero? } f \Rightarrow [0, 0] \\
\text{degree numer } f \geq \text{degree denom } f \Rightarrow \text{error } "\text{Not a proper fraction}" \\
\text{order(denom } f, \text{monomial}(1,1)) \neq 0 \Rightarrow \text{error } "\text{Not integral at } t = 0" \\
r := \text{HermiteIntegrate}(f, \text{derivation}) \\
\text{zero? } g \Rightarrow \\
r.\text{logpart} \neq 0 \Rightarrow "\text{failed}"
\[r.\text{answer}, 0\]
\text{degree numer } g \geq \text{degree denom } g \Rightarrow \text{error } "\text{Not a proper fraction}" \\
\text{order(denom } g, \text{monomial}(1,1)) \neq 0 \Rightarrow \text{error } "\text{Not integral at } t = 0" \\
differentiate(c := r.\text{logpart} / g, \text{derivation}) \neq 0 \Rightarrow "\text{failed}"
\[r.\text{answer}, c\]

\[
\text{limitedLogs}(f, \text{logderiv}, \text{lu}) = \\
\text{zero? } f \Rightarrow \text{empty()} \\
\text{empty? } \text{lu} \Rightarrow "\text{failed}"
\text{empty? rest } \text{lu} \Rightarrow \\
\text{logderiv(c0 := } f / \text{logderiv(u0 := first } \text{lu}) \text{) \neq 0 \Rightarrow "failed}"
[[c0, u0]]
\text{num := numer } f \\
\text{den := denom } f \\
l1: \text{List Record}(\text{logand2:RF}, \text{contrib:UP}) := \\
[[u, \text{num} v] \text{ for } u \text{ in } \text{lu} \mid \text{one? } \text{denom}(v := \text{den} * \text{logderiv } u)]
[[u, \text{num} v] \text{ for } u \text{ in } \text{lu} \mid (\text{denom}(v := \text{den} * \text{logderiv } u) = 1)]
\text{rows := max} (\text{degree } \text{den}, \\
1 + \text{reduce}(\text{max, [degree(u.contrib) for } u \text{ in } \text{l1}])\) \\
\text{0$\text{List(N)}$} \\
m: \text{Matrix(F)} := \text{zero(rows, cols := 1 + } \#\text{l1}) \\
\text{for } i \text{ in } 0...\text{rows-1} \text{ repeat}
\text{for pp in l1 for } j \text{ in minColIndex } m .. \text{maxColIndex } m - 1 \text{ repeat}
\text{qsetelt!}(m, i + \text{minRowIndex } m, j, \text{coefficient(pp.contrib, i)})
\text{qsetelt!}(m, i+\text{minRowIndex } m, \text{maxColIndex } m, \text{coefficient(num, i)})
\text{m := rowEchelon } m \\
\text{ans := empty()$\text{LLG}$} \\
\text{for } i \text{ in } \text{minRowIndex } m .. \text{maxRowIndex } m \mid \\
\text{qelt}(m, i, \text{maxColIndex } m) \neq 0 \text{ repeat}
\text{OK := false} \\
\text{for } pp \text{ in l1 for } j \text{ in minColIndex } m .. \text{maxColIndex } m - 1 \text{ while not } \text{OK} \text{ repeat}
\text{if } \text{qelt}(m, i, j) \neq 0 \text{ then}
\text{OK := true} \\
c := \text{qelt}(m, i, \text{maxColIndex } m) / \text{qelt}(m, i, j) \\
\text{logderiv(c0 := } c::\text{UP::RF} \text{) \neq 0 \Rightarrow return } "\text{failed}"
\text{ans := concat([c0, pp.logand2], ans)}
\text{not } \text{OK} \Rightarrow \text{return } "\text{failed}"
\text{ans}

--- returns \( q \) in UP s.t. \( p = q' \), or "failed"

\[
\text{primntflidpoly}(p, \text{extendedint}, t') =
\]
(u := primintegratepoly(p, extendedint, t')) case UPUP => "failed"
u.a0 ^= 0 => "failed"
u.answer

-- returns q in GP st p = q', or "failed"
expintf1dpoly(p, FRDE) ==
  (u := expintegratepoly(p, FRDE)) case GPGP => "failed"
u.a0 ^= 0 => "failed"
u.answer

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + /[ci * ui'/ui]
-- and a = 0 or a has no integral in UP
primlimitedint(f, derivation, extendedint, lu) ==
  qr := divide(numer f, denom f)
  (u1 := primlimintfrac(qr.remainder / (denom f), derivation, lu))
  case "failed" => "failed"
  (u2 := primintegratepoly(qr.quotient, extendedint, retract derivation monomial(1, 1))) case UPUP => "failed"
  [[u1.mainpart + u2.answer::RF, u1.limitedlogs], u2.a0]

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + /[ci * ui'/ui]
-- and a = 0 or a has no integral in F
explimitedint(f, derivation, FRDE, lu) ==
  qr := separate(f)$GP
  (u1 := explimintfrac(qr.fracPart, derivation, lu)) case "failed" =>
    "failed"
  (u2 := expintegratepoly(qr.polyPart, FRDE)) case GPGP => "failed"
  [[u1.mainpart + convert(u2.answer)@RF, u1.limitedlogs], u2.a0]

-- returns [v, c1...cn] s.t. f = v' + /[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
primlimintfrac(f, derivation, lu) ==
  zero? f => [0, empty()]
  degree numer f >= degree denom f => error "Not a proper fraction"
  r := HermiteIntegrate(f, derivation)
  zero?(r.logpart) => [r.answer, empty()]
  (u := limitedLogs(r.logpart, x1 +-> logprmderiv(x1, derivation), lu))
  case "failed" => "failed"
  [r.answer, u::LLG]

-- returns [v, c1...cn] s.t. f = v' + /[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
-- f must be integral above t = 0
explimintfrac(f, derivation, lu) ==
  zero? f => [0, empty()]
  degree numer f >= degree denom f => error "Not a proper fraction"
  order(denom f, monomial(1,1)) > 0 => error "Not integral at t = 0"
  r := HermiteIntegrate(f, derivation)
zero?(r.logpart) => [r.answer, empty()]
eta' := coefficient(derivation monomial(1, 1), 1)
(u := limitedLogs(r.logpart, x1 +-> logexpderiv(x1, derivation, eta'), lu))
  case "failed" => "failed"
[r.answer - eta'::UP *
  /[((degree numer(v.logand))::Z - (degree denom(v.logand))::Z) *
    v.coef for v in u], u::LLG]

logexpderiv(f, derivation, eta') ==
(differentiate(f, derivation) / f) -
  (((degree numer f)::Z - (degree denom f)::Z) * eta')::UP::RF

notelementary rec ==
  rec.ir + integral(rec.polypart::RF + rec.specpart, monomial(1,1)$SUP :: RF)
-- returns
-- (g in IR, a in F) st f = g' + a, and a=0 or a has no integral in UP
primintegrate(f, derivation, extendedint) ==
  rec := monomialIntegrate(f, derivation)
  not elem?(i1 := rec.ir) => [notelementary rec, 0]
  (u2 := primintegratepoly(rec.polypart, extendedint,
    retract derivation monomial(1, 1)) case UPUUP =>
    [i1 + u2.elem::RF::IR
     + integral(u2.notelem::RF, monomial(1,1)$UP :: RF), 0]
    [i1 + u2.answer::RF::IR, u2.a0]
-- returns
-- (g in IR, a in F) st f = g' + a, and a = 0 or a has no integral in F
expintegrate(f, derivation, FRDE) ==
  rec := monomialIntegrate(f, derivation)
  not elem?(i1 := rec.ir) => [notelementary rec, 0]
  rec.specpart is either 0 or of the form p(t)/t**n
  special := rec.polypart::GP +
    (numer(rec.specpart)::GP exquo denom(rec.specpart)::GP)::GP
  (u2 := expintegratepoly(special, FRDE)) case GPGP =>
    [i1 + convert(u2.elem)@RF::IR + integral(convert(u2.notelem)@RF,
      monomial(1,1)$SUP :: RF), 0]
    [i1 + convert(u2.answer)@RF::IR, u2.a0]
-- returns
-- (g in IR, a in F) st f = g' + a, and a = 0 or a has no integral in F
tanintegrate(f, derivation, FRDE) ==
  rec := monomialIntPoly(rec.polypart, derivation)
  t := monomial(1, 1)$SUP
c := coefficient(r.polypart, 1) / leadingCoefficient(derivation t)
derivation(c::UP) ^= 0 =>
    [i1 + mkAnswer(r.answer::RF, empty(),
      [r.polypart::RF + rec.specpart, dummy]$NE]), 0]
logs:List(LOG) :=
    zero? c => empty()
    [[1, monomial(1,1)::UPRF - (c/(2::F))::UPRF, (1 + t**2)::RF::UPR]]
c0 := coefficient(r.polypart, 0)
(u := tanintegratespecial(rec.specpart, x->differentiate(x, derivation),
FRDE)) case RFRF =>
    [i1 + mkAnswer(r.answer::RF + u.elem, logs, 
     [[u.notelem,dummy]$NE]), c0]
    [i1 + mkAnswer(r.answer::RF + u.answer, logs, empty()), u.a0 + c0]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- or (v in RF, a in F) s.t. f = v' + a
-- and a = 0 or a has no integral in UP
primextendedint(f, derivation, extendedint, g) =>
    fqr := divide(numer f, denom f)
gqr := divide(numer g, denom g)
(u1 := primextintfrac(fqr.remainder / (denom f), derivation,
   gqr.remainder / (denom g))) case "failed" => "failed"
    zero?(gqr.remainder) =>
        -- the following FAIL cannot occur if the primitives are all logs
degree(gqr.quotient) > 0 => FAIL
        (u3 := primintegratepoly(fqr.quotient, extendedint,
         retract derivation monomial(1, 1))) case UPUP => "failed"
        [u1.ratpart + u3.answer::RF, u3.a0]
    (u2 := primintfldpoly(fqr.quotient - retract(u1.coeff)@UP *
gqr.quotient, extendedint, retract derivation monomial(1, 1)))
        case "failed" => "failed"
        [u2::UP::RF + u1.ratpart, u1.coeff]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- or (v in RF, a in F) s.t. f = v' + a
-- and a = 0 or a has no integral in F
expextendedint(f, derivation, FRDE, g) =>
    qf := separate(f)$GP
    qg := separate g
    (u1 := expextintfrac(qf.fracPart, derivation, qg.fracPart))
        case "failed" => "failed"
    zero?(qg.fracPart) =>
        -- the following FAIL's cannot occur if the primitives are all logs
        retractIfCan(qg.polyPart)@Union(F,"failed") case "failed"=> FAIL
        (u3 := expintegratepoly(qf.polyPart,FRDE)) case GGP => "failed"
        [u1.ratpart + convert(u3.answer)$RF, u3.a0]
    (u2 := expintfldpoly(qf.polyPart - retract(u1.coeff)@UP :: GP
 * qg.polyPart, FRDE)) case "failed" => "failed"
        [convert(u2::GP)$RF + u1.ratpart, u1.coeff]

-- returns either
-- (q in UP, a in F) st p = q' + a, and a=0 or a has no integral in UP
-- or (q in UP, r in UP) st p = q' + r, and r has no integral elem/UP
primintegratepoly(p, extendedint, t') =>
    zero? p => [0, 0$F]
ans:UP := 0

while (d := degree p) > 0 repeat
  (ans1 := extendedint leadingCoefficient p) case "failed" =>
    return([ans, p])
  p := reductum p - monomial(d * t' * ans1.ratpart, (d - 1)::N)
  ans := ans + monomial(ans1.ratpart, d)
    + monomial(ans1.coeff / (d + 1)::F, d + 1)
  (ans1 := extendedint(rp := retract(p)$F)) case "failed" => [ans, rp]
  [monomial(ans1.coeff, 1) + ans1.ratpart::UP + ans, 0$F]

-- returns (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- g must have a squarefree denominator (always possible)
-- g must have no polynomial part (degree numer g < degree denom g)
-- f must have no polynomial part (degree numer f < degree denom f)
primextintfrac(f, derivation, g) ==
  zero? f => [0, 0]
  degree numer f >= degree denom f => error "Not a proper fraction"
  r := HermiteIntegrate(f, derivation)
  zero? g =>
    r.logpart ^= 0 => "failed"
    [r.answer, 0]
  degree numer g >= degree denom g => error "Not a proper fraction"
  differentiate(c := r.logpart / g, derivation) ^= 0 => "failed"
  [r.answer, c]

package TRMANIP TranscendentalManipulations

— TranscendentalManipulations.input —

)set break resume
)sys rm -f TranscendentalManipulations.output
)spool TranscendentalManipulations.output
)set message test on
)set message auto off
TranscendentalManipulations examples

TranscendentalManipulations provides functions to simplify and expand expressions involving transcendental operators.

See Also:
o )show TranscendentalManipulations

Exports:
cos2sec  cosh2sech  cot2tan
cot2trig  coth2tanh  coth2trigh
csc2sin   csch2sinh  expand
expandLog expandPower htrigs
removeCosSq removeCoshSq removeSinSq
removeSinhSq sec2cos  sech2cosh
simplify  simplifyExp simplifyLog
sin2csc   sinh2csch  tan2cot
tan2trig  tanh2coth  tanh2trigh
expandTrigProducts

— package TRMANIP TranscendentalManipulations —

)abbrev  package TRMANIP TranscendentalManipulations
++ Author: Bob Sutor, Manuel Bronstein
++ Date Created: Way back
++ Date Last Updated: 22 January 1996, added simplifyLog MCD.
++ Description:
++ TranscendentalManipulations provides functions to simplify and
++ expand expressions involving transcendental operators.

TranscendentalManipulations(R, F): Exports == Implementation where
R : Join(OrderedSet, GcdDomain)
F : Join(FunctionSpace R, TranscendentalFunctionCategory)

Z   ==>   Integer
K   ==>   Kernel F
P   ==>   SparseMultivariatePolynomial(R, K)
UP  ==>   SparseUnivariatePolynomial P
POWER ==> "%power"::Symbol
POW  ==>   Record(val: F, exponent: Z)
PRODUCT ==> Record(coef : Z, var : K)
FPR  ==>   Fraction Polynomial R

Exports ==> with
  expand  : F -> F
    ++ expand(f) performs the following expansions on f:
    ++ \item 1. logs of products are expanded into sums of logs,
    ++ \item 2. trigonometric and hyperbolic trigonometric functions
    ++ of sums are expanded into sums of products of trigonometric
    ++ and hyperbolic trigonometric functions.
    ++ \item 3. formal powers of the form \spad{(a/b)**c} are expanded into
    ++ \spad{a**c * b**(-c)}.
    ++ \end(items)

  simplify  : F -> F
    ++ simplify(f) performs the following simplifications on f:
    ++ \item 1. rewrites trigs and hyperbolic trigs in terms
    ++ of \spad{sin}, \spad{cos}, \spad{sinh}, \spad{cosh}.
    ++ \item 2. rewrites \spad{sin**2} and \spad{sinh**2} in terms
++ of \spad{cos} and \spad{cosh},
++ \item 3. rewrites \spad{exp(a)*exp(b)} as \spad{exp(a+b)}.
++ \item 4. rewrites \spad{(a**(1/n))**m * (a**(1/s))**t} as a single
++ power of a single radical of \spad{a}.
++ \end{items}

htrigs : F -> F
++ htrigs(f) converts all the exponentials in f into
++ hyperbolic sines and cosines.
simplifyExp: F -> F
++ simplifyExp(f) converts every product \spad{exp(a)*exp(b)}
++ appearing in f into \spad{exp(a+b)}.
simplifyLog : F -> F
++ simplifyLog(f) converts every \spad{log(a) - log(b)} appearing in f
++ into \spad{log(a/b)}, every \spad{log(a) + log(b)} into \spad{log(a*b)}
++ and every \spad{n*log(a)} into \spad{log(a^n)}.
expandPower: F -> F
++ expandPower(f) converts every power \spad{(a/b)**c} appearing
++ in f into \spad{a**c * b**(-c)}.
expandLog : F -> F
++ expandLog(f) converts every \spad{log(a/b)} appearing in f into
++ \spad{log(a) - log(b)}, and every \spad{log(a*b)} into
++ \spad{log(a) + log(b)}.

++ cos2sec : F -> F
++ cos2sec(f) converts every \spad{cos(u)} appearing in f into
++ \spad{1/sec(u)}.

++ cosh2sech : F -> F
++ cosh2sech(f) converts every \spad{cosh(u)} appearing in f into
++ \spad{1/sech(u)}.

cot2trig : F -> F
++ cot2trig(f) converts every \spad{cot(u)} appearing in f into
++ \spad{cos(u)/sin(u)}.

++ coth2trigh : F -> F
++ coth2trigh(f) converts every \spad{coth(u)} appearing in f into
++ \spad{cosh(u)/sinh(u)}.

csc2sin : F -> F
++ csc2sin(f) converts every \spad{csc(u)} appearing in f into
++ \spad{1/sin(u)}.

++ csch2sinh : F -> F
++ csch2sinh(f) converts every \spad{csch(u)} appearing in f into
++ \spad{1/sinh(u)}.

sec2cos : F -> F
++ sec2cos(f) converts every \spad{sec(u)} appearing in f into
++ \spad{1/cos(u)}.

++ sech2cosh : F -> F
++ sech2cosh(f) converts every \spad{sech(u)} appearing in f into
++ \spad{1/cosh(u)}.

++ sin2csc : F -> F
++ sin2csc(f) converts every \spad{sin(u)} appearing in f into
++ \spad{1/csc(u)}.

++ sinh2csch : F -> F
++ sinh2csch(f) converts every \spad{sinh(u)} appearing in f into
++ \spad{1/csch(u)}.

\textbf{tan2trig} : F -> F
++ tan2trig(f) converts every \spad{tan(u)} appearing in f into
++ \spad{sin(u)/cos(u)}.
\textbf{tanh2trigh} : F -> F
++ tanh2trigh(f) converts every \spad{tanh(u)} appearing in f into
++ \spad{sinh(u)/cosh(u)}.

\textbf{tan2cot} : F -> F
++ tan2cot(f) converts every \spad{tan(u)} appearing in f into
++ \spad{1/cot(u)}.

\textbf{tanh2coth} : F -> F
++ tanh2coth(f) converts every \spad{tanh(u)} appearing in f into
++ \spad{1/coth(u)}.

\textbf{cot2tan} : F -> F
++ cot2tan(f) converts every \spad{cot(u)} appearing in f into
++ \spad{1/tan(u)}.

\textbf{coth2tanh} : F -> F
++ coth2tanh(f) converts every \spad{coth(u)} appearing in f into
++ \spad{1/tanh(u)}.

\textbf{removeCosSq} : F -> F
++ removeCosSq(f) converts every \spad{cos(u)**2} appearing in f into
++ \spad{1 - sin(x)**2}, and also reduces higher
++ powers of \spad{cos(u)} with that formula.

\textbf{removeSinSq} : F -> F
++ removeSinSq(f) converts every \spad{sin(u)**2} appearing in f into
++ \spad{1 - cos(x)**2}, and also reduces higher powers of
++ \spad{sin(u)} with that formula.

\textbf{removeCoshSq} : F -> F
++ removeCoshSq(f) converts every \spad{cosh(u)**2} appearing in f into
++ \spad{1 - sinh(x)**2}, and also reduces higher powers of
++ \spad{cosh(u)} with that formula.

\textbf{removeSinhSq} : F -> F
++ removeSinhSq(f) converts every \spad{sinh(u)**2} appearing in f into
++ \spad{1 - cosh(x)**2}, and also reduces higher powers
++ of \spad{sinh(u)} with that formula.

if R has PatternMatchable(R) and R has ConvertibleTo(Pattern(R))
and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then

\textbf{expandTrigProducts} : F -> F
++ expandTrigProducts(e) replaces \axiom{sin(x)*sin(y)} by
++ \spad{(cos(x-y)-cos(x+y))/2}, \axiom{cos(x)*cos(y)} by
++ \spad{(cos(x-y)+cos(x+y))/2}, and \axiom{sin(x)*cos(y)} by
++ \spad{(sin(x-y)+sin(x+y))/2}. Note that this operation uses
++ the pattern matcher and so is relatively expensive. To avoid
++ getting into an infinite loop the transformations are applied
++ at most ten times.

Implementation ==> add
import FactoredFunctions(P)
import PolynomialCategoryLifting(IndexedExponents K, K, R, P, F)
import PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,P,F)

smexp : P -> F
termexp : P -> F
exlog : P -> F
smlog : P -> F
smp2htrigs : P -> F
kerexpand : K -> F
expandpow : K -> F
logexpand : K -> F
sup2htrigs : (UP, F) -> F
supexp : (UP, F, F, Z) -> F
ueval : (F, String, F -> F) -> F
ueval2 : (F, String, F -> F) -> F
powersimp : (P, List K) -> F
t2t : F -> F
c2t : F -> F
c2s : F -> F
s2c : F -> F
s2c2 : F -> F
th2th : F -> F
ch2th : F -> F
ch2sh : F -> F
sh2ch : F -> F
sh2ch2 : F -> F
simplify0 : F -> F
simplifyLog1 : F -> F
logArgs : List F -> F

import F
import List F

if R has PatternMatchable R and R has ConvertibleTo Pattern R
and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then
  XX : F := coerce new()$Symbol
  YY : F := coerce new()$Symbol
  sinCosRule : RewriteRule(R,R,F) :=
    rule(cos(XX)*sin(YY),(sin(XX+YY)-sin(XX-YY))/2::F)
  sinSinRule : RewriteRule(R,R,F) :=
    rule(sin(XX)*sin(YY),(cos(XX-YY)-cos(XX+YY))/2::F)
  cosCosRule : RewriteRule(R,R,F) :=
    rule(cos(XX)*cos(YY),(cos(XX-YY)+cos(XX+YY))/2::F)
  sinSum : RewriteRule(R,R,F) :=
    rule(sin(XX+YY),((sin(XX)*cosh(YY)+cosh(XX)*sinh(YY))::F)
  coshSum : RewriteRule(R,R,F) :=
    rule(cosh(XX+YY),((cosh(XX)*cosh(YY)+sinh(XX)*sinh(YY))::F)
  tanhSum : RewriteRule(R,R,F) :=
    rule(tanh(XX+YY),((tanh(XX)+tanh(YY))/((1+tanh(XX)*tanh(YY)))::F)
cothSum : RewriteRule(R,R,F) :=
  rule(coth(XX+YY),((coth(XX)*coth(YY)+1)/(coth(YY)+coth(XX)))::F)
sinhpsinh : RewriteRule(R,R,F) :=
  rule(sinh(XX)+sinh(YY),(2*sinh(1/2*(XX+YY))*cosh(1/2*(XX-YY)))::F)
sinhsinh : RewriteRule(R,R,F) :=
  rule(sinh(XX)-sinh(YY),(2*cosh(1/2*(XX+YY))*sinh(1/2*(XX-YY)))::F)
coshpcosh : RewriteRule(R,R,F) :=
  rule(cosh(XX)+cosh(YY),(2*cosh(1/2*(XX+YY))*cosh(1/2*(XX-YY)))::F)
coshmcosh : RewriteRule(R,R,F) :=
  rule(cosh(XX)-cosh(YY),(2*sinh(1/2*(XX+YY))*sinh(1/2*(XX-YY)))::F)
expandTrigProducts(e:F):F ==
  applyRules([sinCosRule,sinSinRule,cosCosRule,
    sinhSum,coshSum,tanhSum,cothSum,
    sinhpsinh,sinhsinh,coshpcosh,
    coshmcosh],e,10)$ApplyRules(R,R,F)

logArgs(l:List F):F ==
  -- This function will take a list of Expressions (implicitly a sum) and
  -- add them up, combining log terms. It also replaces n*log(x) by
  -- log(x^n).
  import K
  sum : F := 0
  arg : F := 1
  for term in l repeat
    is?(term,"log"::Symbol) =>
      arg := arg * simplifyLog(first(argument(first(kernels(term)))))
    -- Now look for multiples, including negative ones.
    prod : Union(PRODUCT, "failed") := isMult(term)
    (prod case PRODUCT) and is?(prod.var,"log"::Symbol) =>
      arg := arg * simplifyLog ((first argument(prod.var))**(prod.coef))
    sum := sum+term
  sum+log(arg)
simplifyLog(e:F):F ==
  simplifyLog1(numerator e)/simplifyLog1(denominator e)
simplifyLog1(e:F):F ==
  freeOf?(e,"log"::Symbol) => e
  -- Check for n*log(u)
  prod : Union(PRODUCT, "failed") := isMult(e)
  (prod case PRODUCT) and is?(prod.var,"log"::Symbol) =>
    log simplifyLog ((first argument(prod.var))**(prod.coef))
  termList : Union(List(F),"failed") := isTimes(e)
  -- I'm using two variables, termList and terms, to work round a
  -- bug in the old compiler.
  not (termList case "failed") =>
    -- We want to simplify each log term in the product and then multiply
    -- them together. However, if there is a constant or arithmetic
-- expression (i.e. something which looks like a Polynomial) we would
-- like to combine it with a log term.
terms : List F := [simplifyLog(term) for term in termList::List(F)]
exprs : List F := []
for i in 1..#terms repeat
  if retractIfCan(terms.i)@Union(FPR, "failed") case FPR then
    exprs := cons(terms.i, exprs)
    terms := delete!(terms, i)
  if not empty? exprs then
    foundLog := false
    i : NonNegativeInteger := 0
    while (not(foundLog) and (i < #terms)) repeat
      i := i + 1
      if is?(terms.i, "log"::Symbol) then
        args : List F := argument(retract(terms.i)@K)
        setelt(terms.i, log simplifyLog1(first(args)**(*/exprs)))
        foundLog := true
      endif
    endwhile
  endif
  if not foundLog then terms := append(exprs, terms)
endfor
*/terms

terms : Union(List(F), "failed") := isPlus(e)
not (terms case "failed") => logArgs(terms)

expt : Union(POW, "failed") := isPower(e)
-- (expt case POW) and not one? expt.exponent =>
(expt case POW) and not (expt.exponent = 1) =>
simplifyLog(expt.val)**(expt.exponent)

kers : List K := kernels e
-- not(one?(#kers)) => e -- Have a constant
not(((#kers) = 1)) => e -- Have a constant
kernel(operator first kers, [simplifyLog(u) for u in argument first kers])

if R has RetractableTo Integer then
  simplify x := rootProduct(simplify0 x)$AlgebraicManipulations(R, F)
else simplify x := simplify0 x

expandpow k ==
a := expandPower first(arg := argument k)
b := expandPower second arg
-- ne:F := (one? numer a => 1; numer(a)::F ** b)
ne:F := (((numer a) = 1) => 1; numer(a)::F ** b)
-- de:F := (one? denom a => 1; denom(a)::F ** (-b))
de:F := (((denom a) = 1) => 1; denom(a)::F ** (-b))
ne * de
termexp p ==
exponent:F := 0
coef := (leadingCoefficient p)::F
lpow := select((z:K):Boolean+->is?(z,POWER)$K, lk := variables p)$List(K)
for k in lk repeat
d := degree(p, k)
if is?(k, "exp"::Symbol) then
  exponent := exponent + d * first argument k
else if not is?(k, POWER) then
  -- Expand arguments to functions as well ... MCD 23/1/97
  --coef := coef * monomial(1, k, d)
  coef := coef *
  monomial(1, kernel(operator k, 
    [simplifyExp u for u in argument k], height k), d)
coef::F * exp exponent * powersimp(p, lpow)

expandPower f ==
l := select((z:K):Boolean +-> is?(z, POWER)$K, kernels f)$List(K)
eval(f, l, [expandpow k for k in l])

-- l is a list of pure powers appearing as kernels in f
powersimp(p, l) ==
empty? l => 1
k := first l -- k = a**b
a := first(arg := argument k)
exponent := degree(p, k) * second arg
empty?(lk := select((z:K):Boolean +-> a = first argument z, rest l)) =>
  (a ** exponent) * powersimp(p, rest l)
for k0 in lk repeat
  exponent := exponent + degree(p, k0) * second argument k0
(a ** exponent) * powersimp(p, setDifference(rest l, lk))

  t2t x == sin(x) / cos(x)
c2t x == cos(x) / sin(x)
c2s x == inv sin x
s2c x == inv cos x
s2c2 x == 1 - cos(x)**2
th2th x == sinh(x) / cosh(x)
ch2th x == cosh(x) / sinh(x)
ch2sh x == inv sinh x
sh2ch x == inv cosh x
sh2ch2 x == cosh(x)**2 - 1
ueval(x, s,f) == eval(x, s::Symbol, f)
ueval2(x,s,f) == eval(x, s::Symbol, 2, f)
cos2sec x == ueval(x, "cos", (z1:F):F + > inv sec z1)
sin2csc x == ueval(x, "sin", (z1:F):F + > inv csc z1)
csc2sin x == ueval(x, "csc", c2s)
sec2cos x == ueval(x, "sec", s2c)
tan2cot x == ueval(x, "tan", (z1:F):F + > inv cot z1)
The htrigs function

The htrigs function can be used to replace and reduce hyperbolic trigonometric identities. The identity for $\sinh(x)$ is $(e^x - e^{-x})/2$.

If we difference these we should get zero

$$ f := \sinh(x) - (e^x - e^{-x})/2 $$

instead, by default, we get

$$ x -x 
2\sinh(x) - \frac{e^{-x}}{e} + \frac{e^x}{e} 
\frac{-------------}{2} $$

The function htrigs(f) gives 0.

This works as follows:

```math
m := mainKernel f => \sinh(x)
Type: Union(Kernel(Expression(Integer)), ...)
```

which is coerced to the first part of the union:
k:=m::Kernel(Expression(Integer))

and the operator is extracted:

\[
\text{op:=} \text{operator}(k) \Rightarrow \sinh
\]

Type: BasicOperator

The argument function extracts the variable used as arguments:

argument k =>> [x]  
Type Kernel(Expression(Integer))

At this point we have picked apart the main Kernel into its operator and its arguments. We now process the list of arguments.

The function htrigs is called on every element of the argument list, which in this case, returns a list:

\[
\text{arg:=[htrigs x for x in argument k]$List(Expression(Integer))
=> [x]
}\]

Type: List(Expression(Integer))

We form a polynomial by replacing the kernel in the numerator with ?

\[
\text{num := univariate(numer f, k)}
\]

\[
x - x
2? - %e + %e
\]

Type: SparseUnivariatePolynomial(SparseMultivariatePolynomial(Integer, Kernel(Expression(Integer))))

and a polynomial of the denominator, replacing the kernel

\[
\text{den := univariate(denom f, k)}
\]

\[
2
\]

Type: SparseUnivariatePolynomial(SparseMultivariatePolynomial(Integer, Kernel(Expression(Integer))))

In this case the op is not the exponential so we are doing straight trig substitution. We reconstruct the function call using the op and arg values, that is:

\[
\text{g1 := op arg =>> sinh(x)}
\]

Type: Expression(Integer)

So sup2htrigs, which is a local function, is used to simplify the parts of the fraction. In this case,
Thus, the result is 0
The identity for \( \cosh(x) \) is \( (\exp(x) + \exp(-x))/2 \)
If we difference these we should get zero

\[
f := \cosh(x) - (\exp^x + \exp^{-x})/2
\]

instead, by default, we get

\[
\frac{x - x}{2} - \frac{\exp^x + \exp^{-x} + 2\cosh(x)}{2}
\]

and the function call \( htrigs(f) \) gives 0
This works as follows:

\[
m := \text{mainKernel } f \Rightarrow \exp^x
\]

which is coerced to the first part of the union:

\[
k := m : \text{Kernel(Expression(Integer))} \Rightarrow \exp^x
\]

and the operator is extracted:

\[
\text{op} := \text{operator}(k) \Rightarrow \exp
\]

The argument function extracts the variable used as arguments:

\[
\text{argument } k \Rightarrow [x]
\]

At this point we have picked apart the main Kernel into its operator and its arguments. We now process the list of arguments.
The \( htrigs \) function is called on every element of the argument list, which in this case, returns a list:
arg:=[htrigs x for x in argument k]$List(Expression(Integer))
  => [x]
  Type: List(Expression(Integer))

We form polynomial by replacing the kernel in the numerator with ?

num := univariate(numer f, k)

- x
- ? - %e + 2cosh(x)

Type: SparseUnivariatePolynomial(
  SparseMultivariatePolynomial(
    Integer, Kernel(Expression(Integer))))

and a polynomial of the denominator, replacing the kernel

den := univariate(denom f, k)

2

Type: SparseUnivariatePolynomial(
  SparseMultivariatePolynomial(
    Integer, Kernel(Expression(Integer))))

In this case, the expression

is?(op, "exp"::Symbol) => true

so we form

a := first arg => x

Type: Expression(Integer)

since we know that

x
cosh(x)+sinh(x) => %e

we can form this use this expression in substitutions

g1 := cosh(a)+sinh(a) => sinh(x)+cosh(x)

Type: Expression(Integer)

since we know that

- x
cosh(x)-sinh(x) => - %e

we can form this use this expression in substitutions
\begin{verbatim}
g2 := cosh(a)-sinh(a) => -sinh(x)+cosh(x)
    Type: Expression(Integer)

b := (degree num)::Integer quo 2 => 0
    Type: NonNegativeInteger

The supexp function is using the g1 and g2 identities to replace exp(x)

\texttt{supexp(num,g1,g2,b) => sinh(x)-cosh(x)-sinh(x)+2cosh(x)-cosh(x)}
    Type: Expression(Integer)

\texttt{supexp(den,g1,g2,b) => 2}
    Type: Expression(Integer)

which is effectively

\texttt{t1/t2 => (sinh(x)-cosh(x)-sinh(x)+2cosh(x)-cosh(x))/2}
    Type: Expression(Integer)

the last form of which can be rearranged as:

\texttt{(sinh(x)-sinh(x) + 2cosh(x)-cosh(x)-cosh(x))/2 => 0}

so the result is 0

package TRMANIP TranscendentalManipulations

\begin{verbatim}
htrigs f ==
    (m := mainKernel f) case "failed" => f
    op := operator(k := m::K)
    arg := [htrigs x for x in argument k]$List(F)
    num := univariate(numer f, k)
    den := univariate(denom f, k)
    is?(op, "exp"::Symbol) =>
        g1 := cosh(a := first arg) + sinh(a)
        g2 := cosh(a) - sinh(a)
        supexp(num,g1,g2,b:= (degree num)::Z quo 2)/supexp(den,g1,g2,b)
        sup2htrigs(num, g1:= op arg) / sup2htrigs(den, g1)
    supexp(p, f1, f2, bse) ==
        ans:F := 0
        while p ^= 0 repeat
            g := htrigs(leadingCoefficient(p)::F)
            if ((d := degree(p)::Z - bse) >= 0) then
                ans := ans + g * f1 ** d
            else ans := ans + g * f2 ** (-d)
            p := reductum p
        ans
\end{verbatim}

\end{verbatim}
sup2htrigs(p, f) ==
(map(smp2htrigs, p)$SparseUnivariatePolynomialFunctions2(P, F)) f

exlog p == +/[r.coef * log(r.logand::F) for r in log squareFree p]

logexpand k ==
nullary?(op := operator k) => k::F
is?(op, "log"::Symbol) =>
exlog(numer(x := expandLog first argument k)) - exlog denom x
op [expandLog x for x in argument k]$List(F)

kerexpand k ==
nullary?(op := operator k) => k::F
is?(op, POWER) => expandpow k
arg := first argument k
is?(op, "sec"::Symbol) => inv expand cos arg
is?(op, "csc"::Symbol) => inv expand sin arg
is?(op, "log"::Symbol) =>
exlog(numer(x := expand arg)) - exlog denom x
num := numer arg
den := denom arg
(b := (reductum num) / den) ^= 0 =>
a := (leadingMonomial num) / den
is?(op, "exp"::Symbol) => exp(expand a) * expand(exp b)
is?(op, "sin"::Symbol) =>
sin(expand a) * expand(cos b) + cos(expand a) * expand(sin b)
is?(op, "cos"::Symbol) =>
cos(expand a) * expand(cos b) - sin(expand a) * expand(sin b)
is?(op, "tan"::Symbol) =>
ta := tan expand a
tb := expand tan b
(ta + tb) / (1 - ta * tb)
is?(op, "cot"::Symbol) =>
cta := cot expand a
ctb := expand cot b
(cta * ctb - 1) / (ctb + cta)
op [expand x for x in argument k]$List(F)
op [expand x for x in argument k]$List(F)

smpexp p ==
ans:F := 0
while p ^= 0 repeat
  ans := ans + termexp leadingMonomial p
  p := reductum p
ans

-- this now works in 3 passes over the expression:
-- pass1 rewrites trigs and htrigs in terms of sin,cos,sinh,cosh
-- pass2 rewrites sin**2 and sinh**2 in terms of cos and cosh.
-- pass3 groups exponentials together
simplify0 x ==
  simplifyExp eval(eval(x,
    ["tan":Symbol,"cot":Symbol,"sec":Symbol,"csc":Symbol,
    "tanh":Symbol,"coth":Symbol,"sech":Symbol,"csch":Symbol],
    [t2t,c2t,s2c,c2s,th2th,ch2th,sh2ch,ch2sh]),
    ["sin":Symbol, "sinh":Symbol], [2,2], [s2c2, sh2ch2])
Risch differential equation, transcendental case.

See Also:

o )show TranscendentalRischDE

---

TranscendentalRischDE (RDETR)

Exports:

baseRDE  monomRDE  polyRDE

--- package RDETR TranscendentalRischDE ---

)abbrev package RDETR TranscendentalRischDE
++ Author: Manuel Bronstein
++ Date Created: Jan 1988
++ Date Last Updated: 2 November 1995
++ Description:
++ Risch differential equation, transcendental case.

TranscendentalRischDE(F, UP): Exports == Implementation where
F : Join(Field, CharacteristicZero, RetractableTo Integer)
UP : UnivariatePolynomialCategory F

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
REC ==> Record(a:UP, b:UP, c:UP, t:UP)
SPE ==> Record(b:UP, c:UP, m:Z, alpha:UP, beta:UP)
PSOL==> Record(ans:UP, nosol:Boolean)
ANS ==> Union(ans:PSOL, eq:SPE)
PSQ ==> Record(ans:RF, nosol:Boolean)

Exports ==> with
monomRDE: (RF,RF,UP->UP) -> Union(Record(a:UP,b:RF,c:RF,t:UP), "failed")
  ++ monomRDE(f,g,D) returns \spad{[[A, B, C, T]]} such that
  ++ \spad{y' + f y = g} has a solution if and only if \spad{y = Q / T},
  ++ where Q satisfies \spad{A Q' + B Q = C} and has no normal pole.
  ++ A and T are polynomials and B and C have no normal poles.
  ++ D is the derivation to use.
baseRDE : (RF, RF) -> PSQ
  ++ baseRDE(f, g) returns a \spad{[y, b]} such that \spad{y' + fy = g}
  ++ if \spad{b = true}, y is a partial solution otherwise (no solution
  ++ in that case).
  ++ D is the derivation to use.
polyRDE : (UP, UP, UP, Z, UP -> UP) -> ANS
  ++ polyRDE(a, B, C, n, D) returns either:
  ++ 1. \spad{[Q, b]} such that \spad{degree(Q) <= n} and
     ++ \spad{a Q' + B Q = C} if \spad{b = true}, Q is a partial solution
     ++ otherwise.
  ++ 2. \spad{[B1, C1, m, \alpha, \beta]} such that any polynomial solution
     ++ of degree at most n of \spad{A Q' + BQ = C} must be of the form
     ++ \spad{Q = \alpha H + \beta} where \spad{degree(H) <= m} and
     ++ H satisfies \spad{H' + B1 H = C1}.
  ++ D is the derivation to use.
Implementation ==> add
  import MonomialExtensionTools(F, UP)
  getBound : (UP, UP, Z) -> Z
  SPDEnocancel1: (UP, UP, Z, UP -> UP) -> PSOL
  SPDEnocancel2: (UP, UP, Z, Z, F, UP -> UP) -> ANS
  SPDE : (UP, UP, UP, Z, UP -> UP) -> Union(SPE, "failed")

-- cancellation at infinity is possible, A is assumed nonzero
-- needs tagged union because of branch choice problem
-- always returns a PSOL in the base case (never a SPE)
polyRDE(aa, bb, cc, d, derivation) ==
n:Z
  (u := SPDE(aa, bb, cc, d, derivation)) case "failed" => [[0, true]]
  zero?(u.c) => [[u.beta, false]]
-- baseCase? := one?(dt := derivation monomial(1, 1))
baseCase? := ((dt := derivation monomial(1, 1)) = 1)
  n := degree(dt)::Z - 1
  b0? := zero?(u.b)
  (~b0?) and (baseCase? or degree(u.b) > max(0, n)) =>
    answ := SPDEnocancel1(u.b, u.c, u.m, derivation)
    [[u.alpha * answ.ans + u.beta, answ.nosol]]
  (n > 0) and (b0? or degree(u.b) < n) =>
    answ := SPDEnocancel2(u.b,u.c,u.m,n,leadingCoefficient dt,derivation)
    answ case ans=>[[u.alpha * answ.ans.ans + u.beta, answ.ans.nosol]]
\[\begin{align*}
\text{SPDE}(aa, bb, cc, d, \text{derivation}) &= \text{zero? } cc \Rightarrow [0, 0, 0, 0, 0] \\
d < 0 \Rightarrow \text{"failed"} \\
(u := cc \text{ exquo } (g := \text{gcd}(aa, bb))) \text{ case } \text{"failed"} \Rightarrow \text{"failed"} \\
aa := ((aa \text{ exquo } g)::\text{UP}) \\
bb := ((bb \text{ exquo } g)::\text{UP}) \\
cc := u::\text{UP} \\
(ra := \text{retractIfCan}(aa)@\text{Union}(F, \text{"failed"})) \text{ case } F \Rightarrow \\
a1 := \text{inv}(ra::F) \\
[a1 * bb, a1 * cc, d, 1, 0] \\
bc := \text{extendedEuclidean}(bb, aa, cc)::\text{Record}(\text{coef1:UP, coef2:UP}) \\
qr := \text{divide}(bc.\text{coef1}, aa) \\
r := qr.\text{remainder} \\
(v := \text{SPDE}(aa, bb + \text{derivation } aa, \\
bc.\text{coef2} + bb * qr.\text{quotient} - \text{derivation } r, \\
d - \text{degree}(aa)::Z, \text{derivation})) \text{ case } \text{"failed"} \Rightarrow \text{"failed"} \\
[v.b, v.c, v.m, aa * v.\alpha, aa * v.\beta + r]
\end{align*}\]

-- solves \(q' + b q = c\) with \(\text{deg}(q) \leq d\)
-- case (\(B \neq 0\)) and (\(D = d/dt \text{ or } \text{degree}(B) > \text{max}(0, \text{degree}(Dt) - 1)\))
-- this implies no cancellation at infinity, BQ term dominates
-- returns \([Q, \text{flag}]\) such that \(Q\) is a solution if \(\text{flag}\) is false,
-- a partial solution otherwise.
\[\begin{align*}
\text{SPDEnocancel11}(bb, cc, d, \text{derivation}) &= \\
q::\text{UP} := 0 \\
db := (\text{degree } bb)::Z \\
lb := \text{leadingCoefficient } bb \\
\text{while } cc ^= 0 \text{ repeat} \\
d < 0 \text{ or } (n := (\text{degree } cc)::Z - db) < 0 \text{ or } n > d \Rightarrow \text{return } [q, \text{true}]
\end{align*}\]

-- case (\(t\) is a nonlinear monomial) and (\(B = 0\) or degree(B) < degree(Dt) - 1)
-- this implies no cancellation at infinity, DQ term dominates or degree(Q) = 0
-- dtm1 = degree(Dt) - 1
SPDEnocancel2(bb, cc, d, dtm1, lt, derivation) ==
q:UP := 0
while cc ^= 0 repeat
  d < 0 or (n := (degree cc)::Z - dtm1) < 0 or n > d => return [[q, true]]
  if n > 0 then
    r := monomial((leadingCoefficient cc) / (n * lt), n::N)
    cc := cc - bb * r - derivation r
    d := n - 1
    q := q + r
  else -- n = 0 so solution must have degree 0
    db:N := (zero? bb => 0; degree bb);
    db ^= degree(cc) => return [[q, true]]
    zero? db => return [[bb, cc, 0, 1, q]]
    r := leadingCoefficient(cc) / leadingCoefficient(bb)
    cc := cc - r * bb - derivation(r::UP)
    d := - 1
    q := q + r::UP
  [[q, false]]

monomRDE(f, g, derivation) ==
gg := gcd(d := normalDenom(f,derivation), e := normalDenom(g,derivation))
    tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
    (u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
    [aa, aa * f - (d * derivation tt)::RF, u::UP * e * g, tt]

-- solve y' + f y = g for y in RF
-- assumes that f is weakly normalized (no finite cancellation)
-- base case: F' = 0
baseRDE(f, g) ==
  (u := monomRDE(f, g, differentiate)) case "failed" => [0, true]
  n := getBound(u.a,bb := retract(u.b)@UP,degree(cc := retract(u.c)@UP)::Z)
  v := polyRDE(u.a, bb, cc, n, differentiate).ans
  [v.ans / u.t, v.nosol]

-- return an a bound on the degree of a solution of A P'+ B P = C,A ^= 0
-- cancellation at infinity is possible
-- base case: F' = 0
getBound(a, b, dc) ==
da := (degree a)::Z
zero? b => max(0, dc - da + 1)
  db := (degree b)::Z
da > (db + 1) => max(0, dc - da + 1)
da < (db + 1) => dc - db
  (n := retractIfCan(- leadingCoefficient(b) / leadingCoefficient(a))@Union(Z, "failed")) case Z => max(n::Z, dc - db)
dc - db
package RDETRS TranscendentalRischDES\text{ystem}

\text{--- TranscendentalRischDES\text{ystem.input ---}

\text{)}set break resume
\text{)}sys rm -f TranscendentalRischDES\text{ystem.output}
\text{)}spool TranscendentalRischDES\text{ystem.output}
\text{)}set message test on
\text{)}set message auto off
\text{)}clear all

--S 1 of 1
\text{)}show TranscendentalRischDES\text{ystem}
--E 1

\text{)}spool
\text{)}lisp (bye)

\text{--- TranscendentalRischDES\text{ystem.help ---}

====================================================================
TranscendentalRischDES\text{ystem examples
====================================================================

Risch differential equation system, transcendental case.

See Also:
\text{o )show TranscendentalRischDES\text{ystem

\text{---}
TranscendentalRischDESystem (RDETRS)

Exports:
baseRDEsys  monomRDEsys

— package RDETRS TranscendentalRischDESystem —

)abbrev package RDETRS TranscendentalRischDESystem
++ Author: Manuel Bronstein
++ Date Created: 17 August 1992
++ Date Last Updated: 3 February 1994
++ Description:
++ Risch differential equation system, transcendental case.

TranscendentalRischDESystem(F, UP): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP : UnivariatePolynomialCategory F

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
V ==> Vector UP
U ==> Union(List UP, "failed")
REC ==> Record(z1:UP, z2:UP, r1:UP, r2:UP)

Exports ==> with
  monomRDEsys: (RF, RF, RF, UP -> UP) -> _
    ++ monomRDEsys(f,g1,g2,D) returns \spad{\{A, B, H, C1, C2, T\}} such that
    ++ \spad{\{(y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)\}} has a solution
    ++ if and only if \spad{y1 = Q1 / T, y2 = Q2 / T},
    ++ where \spad{\{(Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)\}}
    ++ D is the derivation to use.
  baseRDEsys: (RF, RF, RF) -> Union(List RF, "failed")
    ++ baseRDEsys(f, g1, g2) returns fractions \spad{\{y_1, y_2\}} such that
    ++ \spad{\{(y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)\}}
++ if \spad{y_1,y_2} exist, "failed" otherwise.

Implementation ==> add
import MonomialExtensionTools(F, UP)
import SmithNormalForm(UP, V, V, Matrix UP)

diophant: (UP, UP, UP, UP, UP) -> Union(REC, "failed")
getBound: (UP, UP, UP, UP, UP) -> Z
SPDEsys : (UP, UP, UP, UP, Z, UP -> UP, (F, F, F, UP, UP, Z) -> U) -> U
DSPDEsys: (F, UP, UP, UP, Z, UP -> UP) -> U
DSPDEmix: (UP, UP, F, F, N, Z, F) -> U
DSPDEhdom: (UP, UP, F, F, N, Z) -> U
DSPDEbdom: (UP, UP, F, F, N, Z) -> U
DSPDEsys0: (F, UP, UP, UP, UP, F, F, Z, UP -> UP, (UP,UP,F,F,N) -> U) -> U

-- reduces \((y_1', y_2') + ((0, -f), (f, 0)) (y_1,y_2) = (g_1,g_2)\) to
-- \(A (Q_1', Q_2') + ((H, -B), (B, H)) (Q_1,Q_2) = (C_1,C_2)\), \(Q_1 = y_1 T, Q_2 = y_2 T\)
-- where \(A\) and \(H\) are polynomials, and \(B,C_1,C_2,Q_1\) and \(Q_2\) have no normal poles.
-- assumes that \(f\) is weakly normalized (no finite cancellation)
monomRDEsys(f, g1, g2, derivation) ==
gg := gcd(d := normalDenom(f, derivation),
e := lcm(normalDenom(g1,derivation),normalDenom(g2,derivation)))
tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
(u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
[aa, tt * d * f, - d * derivation tt, u::UP * e * g1, u::UP * e * g2, tt]

-- solve \((y_1', y_2') + ((0, -f), (f, 0)) (y_1,y_2) = (g_1,g_2)\) for \(y_1,y_2\) in \(RF\)
-- assumes that \(f\) is weakly normalized (no finite cancellation) and nonzero
-- base case: \(F' = 0\)
baseRDEsys(f, g1, g2) ==
zero? f => error "baseRDEsys: f must be nonzero"
zero? g1 and zero? g2 => [0, 0]
(u := monomRDEsys(f, g1, g2, differentiate)) case "failed" => "failed"
 n := getBound(u.a, bb := retract(u.b), u.h,
cc1 := retract(u.c1), cc2 := retract(u.c2))
(v := SPDEsys(u.a, bb, u.h, cc1, cc2, n, differentiate,
 (z1,z2,z3,z4,z5,z6)) +->
 DSPDEsys(z1, z2::UP, z3::UP, z4, z5, z6, differentiate)))
case "failed" => "failed"
l := v::List(UP)
[first(l) / u.t, second(l) / u.t]

-- solve
-- \(D_1 = A \ Z_1 + B \ R_1 - C \ R_2\)
-- \(D_2 = A \ Z_2 + C \ R_1 + B \ R_2\)
-- i.e. \(((D_1,D_2) = ((A, 0, B, \ -C), (0, A, C, B)) (Z_1, Z_2, R_1, R_2)\)
-- for \(R_1, R_2\) with degree(Ri) < degree(A)
-- assumes \((A,B,C) = (1)\) and \(A\) and \(C\) are nonzero
diophant(a, b, c, d1, d2) ==
 (u := diophantineSystem(matrix [[a,0,b,-c], [0,a,c,b]],

```
```
vector \([d_1, d_2]\).particular) case "failed" => "failed"

\(v := u::V\)
\(q_1 := \text{divide}(v, a)\)
\(q_2 := \text{divide}(v, a)\)
\([v.1 + b * q_1.\text{quotient} - c * q_2.\text{quotient},
    v.2 + c * q_1.\text{quotient} + b * q_2.\text{quotient}, q_1.\text{remainder}, q_2.\text{remainder}]\)

-- solve
-- \(A (Q_1', Q_2') + ((H, -B), (B, H)) (Q_1, Q_2) = (C_1, C_2)\)
-- for polynomials \(Q_1\) and \(Q_2\) with degree \(\leq n\)
-- \(A\) and \(B\) are nonzero
-- cancellation at infinity is possible

\(\text{SPDEsys}(a, b, h, c_1, c_2, n, \text{derivation}, \text{degradation}) ==\)
\(\text{zero? } c_1 \text{ and zero? } c_2 \Rightarrow [0, 0]\)
\(n < 0 \Rightarrow "\text{failed}"\)
\(g := \text{gcd}(a, \text{gcd}(b, h))\)
\(((u_1 := c_1 \text{ exquo } g) \text{ case } "\text{failed}") \text{ or } ((u_2 := c_2 \text{ exquo } g) \text{ case } "\text{failed}") \Rightarrow "\text{failed}"\)
\(a := (a \text{ exquo } g)::UP\)
\(b := (b \text{ exquo } g)::UP\)
\(h := (h \text{ exquo } g)::UP\)
\(c_1 := u_1::UP\)
\(c_2 := u_2::UP\)
\((\text{da := degree } a) > 0 \Rightarrow (u := \text{diophant}(a, h, b, c_1, c_2)) \text{ case } "\text{failed}" \Rightarrow "\text{failed}"\)
\(\text{rec := } u::\text{REC}\)
\(v := \text{SPDEsys}(a, b, h + \text{derivation } a, \text{rec}.z1 - \text{derivation}(\text{rec.r1}),
    \text{rec}.z2 - \text{derivation}(\text{rec.r2}), n - \text{da}::Z, \text{derivation}, \text{degradation})\)
\(v \text{ case } "\text{failed}" \Rightarrow "\text{failed}"\)
\(l := v::\text{List}(\text{UP})\)
\([a * \text{first}(l) + \text{rec.r1}, a * \text{second}(l) + \text{rec.r2}]\)
\(\text{ra := retract}(a)@F\)
\((\text{rb := retractIfCan}(b)@\text{Union}(\text{F}, "\text{failed}")) \text{ case } "\text{failed}") \text{ or } ((\text{rh := retractIfCan}(h)@\text{Union}(\text{F}, "\text{failed}")) \text{ case } "\text{failed}") \Rightarrow\)
\(\text{DSPDEsys}(\text{ra}, b, h, c_1, c_2, n, \text{derivation})\)
\(\text{degradation}(\text{ra}, \text{rb}::F, \text{rh}::F, c_1, c_2, n)\)

-- solve
-- \(A (Q_1', Q_2') + ((H, -B), (B, H)) (Q_1, Q_2) = (C_1, C_2)\)
-- for polynomials \(Q_1\) and \(Q_2\) with degree \(\leq n\)
-- \(A\) and \(B\) are nonzero
-- cancellation at infinity is not possible

\(\text{DSPDEsys}(a, b, h, c_1, c_2, n, \text{derivation}) ==\)
\(\text{bb := degree}(b)::Z\)
\(\text{hh}::Z :=\)
\(\text{zero? } h \Rightarrow 0\)
\(\text{degree}(h)::Z\)
\(\text{lb := leadingCoefficient } b\)
\(\text{lh := leadingCoefficient } h\)
\(\text{bb < hh} =>\)
DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, getlc) ==
  ans1 := ans2 := 0::UP
repeat
  zero? c1 and zero? c2 => return [ans1, ans2]
  n < 0 or (u := getlc(c1, c2, lb, lh, n::N)) case "failed" => return "failed"
  lq := u::List(UP)
  q1 := first lq
  q2 := second lq
  c1 := c1 - a * derivation(q1) - h * q1 + b * q2
  c2 := c2 - a * derivation(q2) - b * q1 - h * q2
  n := n - 1
  ans1 := ans1 + q1
  ans2 := ans2 + q2

DSPDEmix(c1, c2, lb, lh, n, d, det) ==
  rh1:F :=
    zero? c1 => 0
    (d1 := degree(c1)::Z - d) < n => 0
    d1 > n => return "failed"
    leadingCoefficient c1
  rh2:F :=
    zero? c2 => 0
    (d2 := degree(c2)::Z - d) < n => 0
    d2 > n => return "failed"
    leadingCoefficient c2
  q1 := (rh1 * lh + rh2 * lb) / det
  q2 := (rh2 * lh - rh1 * lb) / det
  [monomial(q1, n), monomial(q2, n)]

DSPDEhdom(c1, c2, lb, lh, n, d) ==
  q1:UP :=
    zero? c1 => 0
    (d1 := degree(c1)::Z - d) < n => 0
    d1 > n => return "failed"
    monomial(leadingCoefficient(c1) / lh, n)
  q2:UP :=
    zero? c2 => 0
    (d2 := degree(c2)::Z - d) < n => 0
    d2 > n => return "failed"
    monomial(leadingCoefficient(c2) / lh, n)
DSPDEbdom(c1, c2, lb, lh, n, d) ==
q1:UP :=
  zero? c2 => 0
  (d2 := degree(c2)::Z - d) < n => 0
  d2 > n => return "failed"
  monomial(leadingCoefficient(c2) / lb, n)
q2:UP :=
  zero? c1 => 0
  (d1 := degree(c1)::Z - d) < n => 0
  d1 > n => return "failed"
  monomial(- leadingCoefficient(c1) / lb, n)
[q1, q2]

-- return a common bound on the degrees of a solution of
-- A (Q1’, Q2’) + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2), Q1 = y1 T, Q2 = y2 T
-- cancellation at infinity is possible
-- a and b are nonzero
-- base case: F’ = 0
getBound(a, b, h, c1, c2) ==
da := (degree a)::Z
dc :=
  zero? c1 => degree(c2)::Z
  zero? c2 => degree(c1)::Z
  max(degree c1, degree c2)::Z
hh:Z :=
  zero? h => 0
  degree(h)::Z
db := max(hh, bb := degree(b)::Z)
da < db + 1 => dc - db
da > db + 1 => max(0, dc - da + 1)
bb >= hh => dc - db
(n := retractIfCan(leadingCoefficient(h) / leadingCoefficient(a)
  )@Union(Z, "failed")) case Z => max(n::Z, dc - db)
dc - db
package SOLVETRA TransSolvePackage

— TransSolvePackage.input —

)set break resume
)sys rm -f TransSolvePackage.output
)spool TransSolvePackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 4
solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
--R
--R (1)
--R [phi= 2atan(%phi0) - theta, phi= 2atan(%phi1) - theta,
--R
--R phi =
--R 2
--R *
--R atan
--R ROOT
--R 4 theta 4 4 2 theta 2
--R - 3v tan(--4) + (- 6v + 24g l v)tan(--2)
--R 2
--R +
--R 4 2 2 2
--R - 3v + 24g l v - 48g l
--R *
--R 2
--R %phi1
--R +
--R 4 theta 4 4 2 theta 2
--R - 2v tan(--4) + (- 4v + 16g l v)tan(--2)
--R 2
--R +
--R 4 2 2 2
--R - 2v + 16g l v - 32g l
--R *
--R %phi0
--R +
--R 2 theta 3 2 2 2 theta
--R - 16g l v tan(--4) + (- 16g l v + 64g l)v tan(--2)
--R 2
--R *
--R %phi1
--R +
--R 4 theta 4 4 2 theta 2
\[\begin{align*}
-3v \tan(-\theta) & + (-6v + 24g_1v)\tan(-\theta) \\
+ & \\
4 & 2 2 2 \\
-3v & + 24g_1v - 48g_1 \\
* & \\
2 & \\
\%phi_0 & \\
+ & \\
2 & \theta_3 \\
-16g_1v \tan(-\theta) & \\
+ & \\
2 & 2 2 \theta \\
(-16g_1v + 64g_1)\tan(-\theta) & \\
* & \\
\%phi_0 & \\
+ & \\
4 & 2 4 4 \theta_2 4 \\
(8v + 16g_1v)\tan(-\theta) & + 16v \tan(-\theta) + 8v \\
+ & \\
-16g_1v - 64g_1 & \\
+ & \\
2 & \theta_2 2 \\
(-v \tan(-\theta) - v + 4g_1)\%phi_1 & \\
+ & \\
2 & \theta_2 2 \\
(-v \tan(-\theta) - v + 4g_1)\%phi_0 - 8g_1 \tan(-\theta) & \\
/ & \\
2 & \theta_2 2 \\
2 \tan(-\theta) & + 2v - 8g_1 \\
+ & \\
- \theta & \\
, & \\
\%phi = & \\
- & \\
2 & \\
* & \\
\atan & \\
ROOT & \\
4 & \theta_4 4 2 \theta_2 \\
-3v \tan(-\theta) & + (-6v + 24g_1v)\tan(-\theta) \\
2 & 2
\end{align*}\]
\begin{verbatim}
+ 4 2 2 2
- 3v + 24 g l v - 48 g l
*
2
%phi1
+
4 theta 4
- 2v tan(-----)
2
+
4 2 theta 2 4 2
(- 4v + 16 g l v )tan(-----) - 2v + 16 g l v
2
+
2 2
- 32 g l
*
%phi0
+
2 theta 3
- 16 g l v tan(-----)
2
+
2 2 2 theta
tan(------)
2
*
%phi1
+
4 theta 4 4 2 theta 2
- 3v tan(-----) + (- 6v + 24 g l v )tan(-----)
2
+
4 2 2 2
- 3v + 24 g l v - 48 g l
*
2
%phi0
+
2 theta 3
- 16 g l v tan(-----)
2
+
2 2 2 theta
(- 16 g l v + 64 g l )tan(-----)
2
*
%phi0
+
\end{verbatim}
\[
\begin{align*}
&\text{definingPolynomial } \%\phi_0 \\
&\text{(2)} \\
&\theta_2 4 \quad \theta_2 4 \quad \theta_2 4 \quad \theta_2 4 \\
&\frac{((\%\phi_0 - 2\%\phi_0 + 1)v + (- 4\%\phi_0 - 4)g l)\tan(\text{------})}{2} \\
&\frac{8\%\phi_0 + 8\%\phi_0)g l tan(\text{------}) + (\%\phi_0 - 2\%\phi_0 + 1)v}{2} \\
&\frac{- 4\%\phi_0 - 4\%\phi_0)g l}{2} \\
&\frac{v \tan(\text{------}) + v - 4g l}{2} \\
&\text{Type: Expression(Integer)}
\end{align*}
\]
\[
\begin{align*}
- \theta^2 + \theta^2 \cos^2 \theta + \cos^2 \theta \\
- \cos^2 \theta + \theta^2 \cos \theta + \cos \theta
\end{align*}
\]
TransSolvePackage examples

This package tries to find solutions of equations of type Expression(R).
This means expressions involving transcendental, exponential, logarithmic
and nthRoot functions.

After trying to transform different kernels to one kernel by applying
several rules, it calls zerosOf for the SparseUnivariatePolynomial in
the remaining kernel.

For example the expression
\[ \sin(x) \cdot \cos(x) - 2 \]
will be transformed to
\[ -2 \tan(x/2)^4 - 2 \tan(x/2)^3 - 4 \tan(x/2)^2 + 2 \tan(x/2) - 2 \]
by using the function normalize and then to
\[ -2 \tan(x)^2 + \tan(x) - 2 \]
with help of subsTan. This function tries to express the given function
in terms of \( \tan(x/2) \) to express in terms of \( \tan(x) \).

Other examples are the expressions
\[ \sqrt{x+1} + \sqrt{x+7} + 1 \]
\[ \sqrt{\sin(x)} + 1 \]
\[ \text{solve}(1/2*v^2*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi) \]

definingPolynomial \( \%phi0 \)

definingPolynomial \( \%phi1 \)

See Also:
\( \)show TransSolvePackage
TransSolvePackage (SOLVETRA)

Exports:
   solve

— package SOLVETRA TransSolvePackage —

)abbrev package SOLVETRA TransSolvePackage
++ Author: W. Wiwianka, Martin Rubey
++ Date Created: Summer 1991
++ Date Last Changed: 9/91
++ Description:
++ This package tries to find solutions of equations of type Expression(R).
++ This means expressions involving transcendental, exponential, logarithmic
++ and nthRoot functions.
++ After trying to transform different kernels to one kernel by applying
++ several rules, it calls zerosOf for the SparseUnivariatePolynomial in
++ the remaining kernel.
++ For example the expression \spad{\sin(x)\cos(x)-2} will be transformed to
++ \spad{-2 \tan(x/2)^4 -2 \tan(x/2)^3 -4 \tan(x/2)^2 +2 \tan(x/2) -2}
++ by using the function normalize and then to
++ \spad{-2 \tan(x)^2 + \tan(x) -2}
++ with help of subsTan. This function tries to express the given function
++ in terms of \spad{\tan(x/2)} to express in terms of \spad{\tan(x)}.
++ Other examples are the expressions \spad{sqrt(x+1)+sqrt(x+7)+1} or
++ \spad{sqrt(sin(x))+1}.

TransSolvePackage(R) : Exports == Implementation where
   R : Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
            LinearlyExplicitRingOver Integer, CharacteristicZero)
   I  => Integer
   NNI => NonNegativeInteger
   RE => Expression R
   EQ => Equation
   S  => Symbol
   V  => Variable
Exports == with

solve : RE -> L EQ RE
++ solve(expr) finds the solutions of the equation expr = 0
++ where expr is a function of type Expression(R)
++ with respect to the unique symbol x appearing in eq.
solve : EQ RE -> L EQ RE
++ solve(eq) finds the solutions of the equation eq
++ where eq is an equation of functions of type Expression(R)
++ with respect to the unique symbol x appearing in eq.
solve : (EQ RE , S) -> L EQ RE
++ solve(eq,x) finds the solutions of the equation eq
++ where eq is an equation of functions of type Expression(R)
++ with respect to the unique symbol x.
solve : (RE , S) -> L EQ RE
++ solve(expr,x) finds the solutions of the equation expr = 0
++ of type Expression(R).
++
++X solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
++X definingPolynomial %phi0
++X definingPolynomial %phi1

solve : (L EQ RE, L S) -> L L EQ RE
++ solve(leqs, lvar) returns a list of solutions to the list of
++ equations leqs with respect to the list of symbols lvar.
-- solve : (L EQ RE, L Kernel RE) -> L L EQ RE
-- ++ solve(leqs, lker) returns a list of solutions to the list
-- ++ of equations leqs with respect to the list of kernels lker.

Implementation == add
import ACF
import HomogeneousAggregate(R)
import AlgebraicManipulations(R, RE)
import TranscendentalManipulations(R, RE)
import TrigonometricManipulations(R, RE)
import ElementaryFunctionStructurePackage(R, RE)
import SparseUnivariatePolynomial(R)
import LinearSystemMatrixPackage(RE, Vector RE, Vector RE, Matrix RE)
import TransSolvePackageService(R)
import MultivariateFactorize(K, IndexedExponents K, R, SMP(R, K))
----- Local Function Declarations -----

solveInner : (RE, S) -> L EQ RE
tryToTrans : (RE, S) -> RE
eliminateKernRoot : (RE, K) -> RE
eliminateRoot : (RE, S) -> RE
combineLog : (RE, S) -> RE
testLog : (RE, S) -> Boolean
splitExpr : (RE) -> L RE
buildnexpr : (RE, S) -> L RE
logsumtolog : RE -> RE
logexpp : (RE, RE) -> RE
testRootk : (RE, S) -> Boolean
testkernel : (RE, S) -> Boolean
funcinv : (RE, RE) -> Union(RE,"failed")
testTrig : (RE, S) -> Boolean
testHTrig : (RE, S) -> Boolean
tableXkernels : (RE, S) -> L RE
subsTan : (RE, S) -> RE

-- exported functions

solve(oside: RE) : L EQ RE ==
  zero? oside => error "equation is always satisfied"
  lv := variables oside
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  solve(oside, lv.first)

solve(equ: EQ RE) : L EQ RE ==
  solve(lhs(equ) - rhs(equ))

solve(equ: EQ RE, x:S) : L EQ RE ==
  oneside := lhs(equ) - rhs(equ)
  solve(oneside, x)

  if R has QuotientFieldCategory(Integer) then
    retractIfCan(rhs sol)@Union(Integer,"failed") case "failed" => true
  else
    retractIfCan(rhs sol)@Union(Fraction Integer,"failed") case "failed" => true
  zero? eval(lside, sol) => true
false

\[\text{solve(lside: RE, x:S) : L EQ RE ==} \]
\[\{\text{sol for sol in solveInner(lside, x) | testZero?(lside, sol)}\}\]

\[\text{solveInner(lside: RE, x:S) : L EQ RE ==} \]
\[\text{lside:=eliminateRoot(lside, x)}\]
\[\text{ausgabe1:=tableXkernels(lside, x)}\]

\[X:=\text{new()@Symbol} \]
\[Y:=\text{new()@Symbol::RE} \]
\[\text{ (#ausgabe1) = 1 =>} \]
\[\text{bigX:= (first ausgabe1)::RE} \]
\[\text{eq1:=eval(lside, bigX=(X::RE))} \]
\[\text{-- Type : Expression R} \]
\[f:=\text{univariate(eq1, first kernels (X::RE))} \]
\[\text{-- Type : Fraction SparseUnivariatePolynomial Expression R} \]
\[\text{lfatt:= factors factorPolynomial numer f} \]
\[\text{lr:L RE := "append" /[zerosOf(fatt.factor, x) for fatt in lfatt]} \]
\[\text{-- Type : List Expression R} \]
\[r1:=[\]::L RE \]
\[\text{for i in 1..#lr repeat} \]
\[\text{finv := funcinv(bigX, lr(i))} \]
\[\text{if finv case RE then r1:=cons(finv::RE, r1)} \]
\[\text{bigX_back:=funcinv(bigX, bigX)::RE} \]
\[\text{if not testkernel(bigX_back, x) then} \]
\[\text{if bigX = bigX_back then return []::L EQ RE} \]
\[\text{return} \]
\[\text{"append"/[solve(bigX_back-ri, x) for ri in r1]} \]
\[\text{newlist:=[\]::L EQ RE} \]
\[\text{for i in 1..#r1 repeat} \]
\[\text{elR := eliminateRoot((numer(bigX_back - r1(i))::RE ),x)} \]
\[\text{f:=univariate(elR, kernel(x))} \]
\[\text{-- Type : Fraction SparseUnivariatePolynomial Expression R} \]
\[\text{lfatt:= factors factorPolynomial numer f} \]
\[\text{secondsol:="append" /[zerosOf(ff.factor, x) for ff in lfatt]} \]
\[\text{for j in 1..#secondsol repeat} \]
\[\text{newlist:=cons(x::RE=rootSimp( secondsol(j) ), newlist)} \]

newlist
\[\text{newlside:=tryToTrans(lside, x) ::RE} \]
\[\text{listofkernels:=tableXkernels(newlside, x)} \]
\[\text{(#listofkernels) = 1 => solve(newlside, x)} \]
\[\text{lfacts := factors factor(numer lside)} \]
\[\text{#lfacts > 1 =>} \]
\[\text{sols : L EQ RE := []} \]
\[\text{for frec in lfacts repeat} \]
\[\text{sols := append(solve(frec.factor :: RE, x), sols)} \]
\[\text{sols} \]
\[\text{return []::L EQ RE} \]
-- local functions

-- This function was suggested by Manuel Bronstein as a simpler alternative to normalize.
simplifyingLog(f:RE):RE ==
  (u:=isExpt(f,"exp"::Symbol)) case Record(var:Kernel RE,exponent:Integer) =>
    rec := u::Record(var:Kernel RE,exponent:Integer)
    rec.exponent * first argument(rec.var)
    log f

testkernel(var1:RE,y:S) : Boolean ==
  var1:=eliminateRoot(var1,y)
  listvar1:=tableXkernels(var1,y)
  if (#listvar1 = 1) and (first argument(listvar1(1)) = y::RE) then
    true
  else if #listvar1 = 0 then true
  else false

solveRetract(lexpr:L RE, lvar:L S):Union(L L EQ RE, "failed") ==
  nlexpr := L Fraction Polynomial R := []
  for expr in lexpr repeat
    rf:=Union(Fraction Polynomial R, "failed") := retractIfCan(expr)$RE
    rf case "failed" => return "failed"
    nlexpr := cons(rf, nlexpr)
  radicalSolve(nlexpr, lvar)$RadicalSolvePackage(R)

tryToTrans(lside: RE, x:S) : RE ==
  if testTrig(lside,x) or testHTrig(lside,x) then
    convLside:=simplify(lside)::RE
    resultLside:=convLside
    listConvLside:=tableXkernels(convLside,x)
    if (#listConvLside) > 1 then
      NormConvLside:=normalize(convLside,x)
      resultLside:=subsTan(NormConvLside , x)
  else if testLog(lside,x) then
    numlside:=numer(lside)::RE
    resultLside:=combineLog(numlside,x)
  else
    NormConvLside:=normalize(lside,x)
    resultLside:=NormConvLside
    listConvLside:=tableXkernels(NormConvLside,x)
    if (#listConvLside) > 1 then
      cnormConvLside:=complexNormalize(lside,x)
      resultLside:=cnormConvLside
listcnorm:=tableXkernels(cnornConvLside,x)
if (#listcnorm) > 1 then
  if testLog(cnornConvLside,x) then
    numlside:=numer(cnornConvLside)::RE
    resultLside:=combineLog(numlside,x)
resultLside

subsTan(exprvar:RE,y:S) : RE ==
  Z:=new()@Symbol
  listofkern:=tableXkernels(exprvar,y)
  varkern:=(first listofkern)::RE
  Y:=(numer first argument first (kernels(varkern)))::RE
  test : Boolean := varkern=tan((Y::RE)/(2::RE))::RE
  if not( (#listofkern=1) and test) then
    return exprvar
  fZ:=eval(exprvar,varkern=(Z::RE))
  fN:=(numer fZ)::RE
  f:=univariate(fN, first kernels(Z::RE))
  secondfun:=(-2*(Y::RE)/((Y::RE)**2-1) )::RE
  g:=univariate(secondfun,first kernels(y::RE))
  H:=(new()@Symbol)::RE
  newH:=univariate(H,first kernels(Z::RE))
  result:=decomposeFunc(f,g,newH)
  if not ( result = f ) then
    result1:=result(H::RE)
    resultnew:=eval(result1,H=(( tan((Y::RE))::RE ) ))
  else return exprvar

eliminateKernRoot(var: RE, varkern: K) : RE ==
  X:=new()@Symbol
  var1:=eval(var, (varkern::RE)=(X::RE) )
  var2:=numer univariate(var1, first kernels(X::RE))
  var3:= monomial(1, ( retract( second argument varkern)@I )::NNI)@SUP RE_
  - monomial(first argument varkern, 0::NNI)@SUP RE
  resultvar:=resultant(var2, var3)

eliminateRoot(var:RE, y:S) : RE ==
  var1:=var
  while testRootk(var1,y) repeat
    varlistk1:=tableXkernels(var1,y)
    for i in varlistk1 repeat
      if is?(i, "nthRoot"::S) then
        var1:=eliminateKernRoot(var1,first kernels(i::RE))
    var1

logsumto(log(var:RE) : RE ==
  (listofexpr:isPlus(var)) case "failed" => var
listofexpr := listofexpr :: L RE
listforgcd := [] :: L R
for i in listofexpr repeat
  expcoeff := leadingCoefficient(numer(i))
  listforgcd := cons(expcoeff, listforgcd)
gcdcoeff := gcd(listforgcd) :: RE
newexpr := RE := 0
for i in listofexpr repeat
  exprlist := splitExpr(i :: RE)
  newexpr := newexpr + logexpr(exprlist.2, exprlist.1/gcdcoeff)
kernelofvar := kernels(newexpr)
var2 := 1 :: RE
for i in kernelofvar repeat
  var2 := var2 * (first argument i)
gcdcoeff * log(var2)

testLog(expr :: RE, Z :: S) : Boolean ==
  testList := [log] :: L S
  kernelofexpr := tableXkernels(expr, Z)
  if #kernelofexpr = 0 then
    return false
  for i in kernelofexpr repeat
    if not member?(name(first kernels(i)), testList) or _
      not testkernel((first argument first kernels(i)), Z) then
      return false
  true

splitExpr(expr :: RE) : L RE ==
  lcoeff := leadingCoefficient((numer expr))
  exprwcoeff := expr
  listexpr := isTimes(exprwcoeff)
  if listexpr case "failed" then
    [1 :: RE, expr]
  else
    listexpr := remove_!(lcoeff :: RE, listexpr)
    cons(lcoeff :: RE, listexpr)

buildnexpr(expr :: RE, Z :: S) : L RE ==
  nlist := splitExpr(expr)
  n2list := remove_!(nlist.1, nlist)
  anscoeff := RE := 1
  ansmant := RE := 0
  for i in n2list repeat
    if freeOf?(i :: RE, Z) then
      anscoeff := (i :: RE) * anscoeff
    else
      ansmant := (i :: RE)
  [anscoeff, ansmant * nlist.1]
logexpp(expr1:RE, expr2:RE) : RE ==
log((first argument first kernels(expr1))**expr2)

combineLog(expr:RE,Y:S) : RE ==
exprtable:Table(RE,RE):=table()
(isPlus(expr)) case "failed" => expr
ans:=RE:=0
while expr ^= 0 repeat
  loopexpr:=leadingMonomial(numer(expr))::RE
  if testLog(loopexpr,Y) and (#tableXkernels(loopexpr,Y)=1) then
    expr:=buildnexpr(loopexpr,Y)
    if search(exprp.1,exprtable) case "failed" then
      exprtable.(exprp.1):=0
      exprtable.(exprp.1):= exprtable.(exprp.1) + exprp.2
    else
      ans:=ans+loopexpr
      expr:=(reductum(numer expr))::RE
      ansexpr:=RE:=0
      for i in keys(exprtable) repeat
        ansexpr:=ansexpr + logsumtolog(exprtable.i) * (i::RE)
      ansexpr:=ansexpr + ans
    end if
  end if
end while

testRootk(varlistk:RE,y:S) : Boolean ==
testList:=[nthRoot]::L S
kernelofeqnvar:=tableXkernels(varlistk,y)
if #kernelofeqnvar = 0 then
  return false
for i in kernelofeqnvar repeat
  if member?(name(first kernels(i)),testList) then
    return true
  end if
return false

tableXkernels(evar:RE,Z:S) : L RE ==
kOfvar:=kernels(evar)
listkOfvar:=[]::L RE
for i in kOfvar repeat
  if not freeOf?(i::RE,Z) then
    listkOfvar:=cons(i::RE,listkOfvar)
  end if
listkOfvar

testTrig(eqnvar:RE,Z:S) : Boolean ==
testList:=[sin , cos , tan , cot , sec , csc]::L S
kernelofeqnvar:=tableXkernels(eqnvar,Z)
if #kernelofeqnvar = 0 then
  return false
for i in kernelofeqnvar repeat
  if not member?(name(first kernels(i)),testList) or _
    not testkernel((first argument first kernels(i)) ,Z) then
    return false
  end if
end for
true

testHTrig(eqnvar:RE,Z:S) : Boolean ==
  testList:=[sinh, cosh, tanh, coth, sech, csch]::L S
  kernelofeqnvar:=tableXkernels(eqnvar,Z)
  if #kernelofeqnvar = 0 then
    return false
  for i in kernelofeqnvar repeat
    if not member?(name(first kernels(i)),testList) or
       not testkernel((first argument first kernels(i)),Z) then
      return false
  true

-- Auxiliary local function for use in funcinv.
makeInterval(1:R):C INT F ==
  if R has complex and R has ConvertibleTo(C F) then
    map(interval$INT(F),convert(1)$R)$ComplexFunctions2(F,INT F)
  else
    error "This should never happen"

funcinv(k:RE,l:RE) : Union(RE,"failed") ==
  is?(k, "sin":Symbol) => asin(l)
  is?(k, "cos":Symbol) => acos(l)
  is?(k, "tan":Symbol) => atan(l)
  is?(k, "cot":Symbol) => acot(l)
  is?(k, "sec":Symbol) =>
    l = 0 => "failed" asec(l)
  is?(k, "csc":Symbol) =>
    l = 0 => "failed" acsc(l)
  is?(k, "sinh":Symbol) => asinh(l)
  is?(k, "cosh":Symbol) => acosh(l)
  is?(k, "tanh":Symbol) => atanh(l)
  is?(k, "coth":Symbol) => acoth(l)
  is?(k, "sech":Symbol) => asech(l)
  is?(k, "csch":Symbol) => acsch(l)
  is?(k, "atan":Symbol) => tan(l)
  is?(k, "acot":Symbol) =>
    l = 0 => "failed" cot(l)
  is?(k, "asin":Symbol) => sin(l)
  is?(k, "acos":Symbol) => cos(l)
  is?(k, "asec":Symbol) => sec(l)
  is?(k, "acsc":Symbol) =>
    l = 0 => "failed" csc(l)
  is?(k, "asinh":Symbol) => sinh(l)
  is?(k, "acosh":Symbol) => cosh(l)
is?(k, "atanh":Symbol) => tanh(l)
is?(k, "acoth":Symbol) =>
  l = 0 => "failed"
  coth(l)
is?(k, "asech":Symbol) => sech(l)
is?(k, "acsch":Symbol) =>
  l = 0 => "failed"
  csch(l)
is?(k, "exp":Symbol) =>
  l = 0 => "failed"
simplifyingLog l
is?(k, "log":Symbol) =>
  if R has complex and R has ConvertibleTo(C F) then
    -- We will check to see if the imaginary part lies in [-Pi,Pi)
    ze : Expression C INT F
    ze := map(makeInterval,l)$ExpressionFunctions2(R,C INT F)
    z : Union(C INT F,"failed") := retractIfCan ze
    z case "failed" => exp l
    im := imag z
    fpi : Float := pi()
    (-fpi < inf(im)) and (sup(im) <= fpi) => exp l
    "failed"
  else -- R not Complex or something which doesn't map to Complex Floats
    exp l
  is?(k, "%power":Symbol) =>
    (t:=normalize(l)) = 0 => "failed"
    log t

import SystemSolvePackage(RE)

ker2Poly(k:Kernel RE, lvar:L S):Polynomial RE ==
  member?(nm:=name k, lvar) => nm :: Polynomial RE
  k :: RE :: Polynomial RE

smp2Poly(pol:SMP(R,Kernel RE), lvar:L S):Polynomial RE ==
  map(x +-> ker2Poly(x, lvar),
       y +-> y::RE::Polynomial RE, pol)$PolynomialCategoryLifting(
    IndexedExponents Kernel RE, Kernel RE, R, SMP(R, Kernel RE),
    Polynomial RE)

makeFracPoly(expr:RE, lvar:L S):Fraction Polynomial RE ==
  smp2Poly(numer expr, lvar) / smp2Poly(denom expr, lvar)

makeREpol(pol:Polynomial RE):RE ==
  lvar := variables pol
  lval : List RE := [v::RE for v in lvar]
  ground eval(pol,lvar,lval)

makeRE(frac:Fraction Polynomial RE):RE ==
makeREpol(numer frac)/makeREpol(denom frac)

solve1Pol(pol:Polynomial RE, var: S, sol:L EQ RE):L L EQ RE ==
  repol := eval(makeREpol pol, sol)
  vsols := solve(repol, var)
  [cons(vsol, sol) for vsol in vsols]

solve1Sys(plist:L Polynomial RE, lvar:L S):L L EQ RE ==
  rplist := reverse plist
  rlvar := reverse lvar
  sols := list(append"/[solve1Pol(p,v,sol) for sol in sols]"

The input

solve(sinh(z)=cosh(z),z)

generates the error (reported as bug # 102):

>> Error detected within library code:
  No identity element for reduce of empty list using operation append

— package SOLVETRA TransSolvePackage —

solveList(lexpr:L RE, lvar:L S):L L EQ RE ==
  ans1 := solveRetract(lexpr, lvar)
  not(ans1 case "failed") => ans1 :: L L EQ RE
  lfrac:L Fraction Polynomial RE :=
    [makeFracPoly(expr, lvar) for expr in lexpr]
  trianglist := triangularSystems(lfrac, lvar)
  l := list(append"/[solve1Sys(plist, lvar) for plist in trianglist]
           reduce(append, l, [])"

solve(leqs:L EQ RE, lvar:L S):L L EQ RE ==
  leexpr := [lhs(eq)-rhs(eq) for eq in leqs]
  ans := solveList(leexpr, lvar)

— solve(leqs:L EQ RE, lker:L Kernel RE):L L EQ RE ==

  leexpr := [lhs(eq)-rhs(eq) for eq in leqs]
  lvar := [new()$S for k in lker]
  lval := [kernel v for v in lvar]
  nleexpr := [eval(expr,lker,lval) for expr in leexpr]
  ans := solveList(nleexpr, lvar)
package SOLVESER TransSolvePackageService

---

SOLVESER TRANSSOLVEPACKAGESERVICE

---

-- lker2 := [v::Kernel(RE) for v in lvar]
-- lval2 := [k::RE for k in lker]
-- [[map(eval(#1,lker2,lval2), neq) for neq in sol] for sol in ans]

---

SOLVESER TransSolvePackageService examples

---

This package finds the function func3 where func1 and func2 are given and func1 = func3(func2). If there is no solution then function func1 will be returned.
An example would be

\[ \text{func1} := 8\times^3 + 32\times^2 - 14\times \] ::EXPR INT

and

\[ \text{func2} := 2\times \] ::EXPR INT

convert them via univariate to \( \text{FRAC SUP EXPR INT} \) and then the solution is

\[ \text{func3} := \times^3 + \times^2 - \times \]

of type \( \text{FRAC SUP EXPR INT} \)

See Also:
- )show TransSolvePackageService

TransSolvePackageService (SOLVESER)

Exports:
- decomposeFunc
- unvectorise

— package SOLVESER TransSolvePackageService —

)abbrev package SOLVESER TransSolvePackageService
++ Author: W. Wiwianka 
++ Date Created: Summer 1991
++ Date Last Changed: 9/91
++ Description:
++ This package finds the function func3 where func1 and func2
++ are given and  \( \text{func1} = \text{func3(func2)} \). If there is no solution then
++ function \( \text{func1} \) will be returned.
++ An example would be \( \spad{\text{func1} := 8*X**3 + 32*X^2 - 14*X :: \text{EXPR INT}} \) and
++ \( \spad{\text{func2} := 2*X :: \text{EXPR INT}} \) convert them via univariate
++ to \( \text{FRAC SUP EXPR INT} \) and then the solution is \( \spad{\text{func3} := X**3 + X**2 - X} \)
++ of type \( \text{FRAC SUP EXPR INT} \)

TransSolvePackageService(R) : Exports == Implementation where
R : Join(IntegralDomain, OrderedSet)

RE ==> Expression R
EQ ==> Equation
S ==> Symbol
V ==> Variable
L ==> List
SUP ==> SparseUnivariatePolynomial
ACF ==> AlgebraicallyClosedField()

Exports == with
decomposeFunc : ( Fraction SUP RE , Fraction SUP RE, Fraction SUP RE ) -> Fraction SUP RE
++ decomposeFunc(func1, func2, newvar) returns a function func3 where
++ func1 = func3(func2) and expresses it in the new variable newvar.
++ If there is no solution then func1 will be returned.
unvectorise : ( Vector RE , Fraction SUP RE , Integer ) -> Fraction SUP RE
++ unvectorise(vect, var, n) returns
++ \( \spad{\text{vect}(1) + \text{vect}(2)*\text{var} + \ldots + \text{vect}(n+1)*\text{var}^n} \) where
++ vect is the vector of the coefficients of the polynomial, var
++ the new variable and n the degree.

Implementation == add
import ACF
import TranscendentalManipulations(R, RE)
import ElementaryFunctionStructurePackage(R, RE)
import SparseUnivariatePolynomial(R)
import LinearSystemMatrixPackage(RE,Vector RE,Vector RE,Matrix RE)
import HomogeneousAggregate(R)

---- Local Function Declarations ----
subsSolve : ( SUP RE , NonNegativeInteger , SUP RE, SUP RE, Integer, Fraction SUP RE ) -> Union(SUP RE , "failed")
++ subsSolve(f, degf, g1, g2, m, h)

-- exported functions

unvectorise(vect:Vector RE, var:Fraction SUP RE, n:Integer) : Fraction SUP RE ==
Z := new() @ Symbol
polyvar : Fraction SUP RE := 0
for i in 1..((n+1)::Integer) repeat
  vecti := univariate(vect(i), first kernels(Z::RE))
  polyvar := polyvar + (vecti) * (var) ** ((n-i+1)::NonNegativeInteger)
polyvar

X := new() @ Symbol
f1 := numer(exprf)
f2 := denom(exprf)
g1 := numer(exprg)
g2 := denom(exprg)
degF := max(degree(numer(exprf)), degree(denom(exprf)))
degG := max(degree(g1), degree(g2))
newF1, newF2 := Union(SUP RE, "failed")
N := degF exquo degG
if not (N case "failed") then
  m := N :: Integer
  newF1 := subsSolve(f1, degF, g1, g2, m, newH)
  if f2 = 1 then
    newF2 := 1 :: SUP RE
  else
    newF2 := subsSolve(f2, degF, g1, g2, m, newH)
  if (not (newF1 case "failed")) and (not (newF2 case "failed")) then
    newF := newF1/newF2
  else return exprf
else return exprf

-- local functions

coefficient := new((DegF+1), 1, 0) @ Matrix RE
for i in 0..M repeat
  coefficient := horizConcat(coefficient, (vectorise( (G1**((M-i)::NonNegativeInteger)) * G2**) vec := vectorise(F, DegF+1)
  coefficient := subMatrix(coefficient, 1, (DegF+1), 2, (M+2))
  solvar := solve(coefficient, vec)
  if not (solvar.particular case "failed") then
    solvevarlist := (solvar.particular) :: Vector RE
    resul := numer(unvectorise(solvevarlist, (HH), M))
    resul
  else return "failed"
package TRIMAT TriangularMatrixOperations

--- TriangularMatrixOperations.input ---

)set break resume
)sys rm -f TriangularMatrixOperations.output
)spool TriangularMatrixOperations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TriangularMatrixOperations
--E 1

)spool
)lisp (bye)

---

--- TriangularMatrixOperations.help ---

====================================================================
TriangularMatrixOperations examples
====================================================================

This package provides functions that compute "fraction-free" inverses of upper and lower triangular matrices over a integral domain. By "fraction-free inverses" we mean the following: given a matrix B with entries in R and an element d of R such that d * inv(B) also has entries in R, we return d * inv(B). Thus, it is not necessary to pass to the quotient field in any of our computations.

See Also:
o )show TriangularMatrixOperations

---
Chapter 21. Triangular Matrix Operations (TRIMAT)

Exports:
LowTriBddDenomInv UpTriBddDenomInv

package TRIMAT TriangularMatrixOperations

)abbrev package TRIMAT TriangularMatrixOperations
++ Author: Victor Miller
++ Date Last Updated: 24 Jul 1990
++ Description:
++ This package provides functions that compute "fraction-free"
++ inverses of upper and lower triangular matrices over a integral
++ domain. By "fraction-free inverses" we mean the following:
++ given a matrix B with entries in R and an element d of R such that
++ d * inv(B) also has entries in R, we return d * inv(B). Thus,
++ it is not necessary to pass to the quotient field in any of our
++ computations.

TriangularMatrixOperations(R,Row,Col,M): Exports == Implementation where
R : IntegralDomain
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)

Exports => with

UpTriBddDenomInv: (M,R) -> M
++ UpTriBddDenomInv(B,d) returns M, where
++ B is a non-singular upper triangular matrix and d is an
++ element of R such that \spad{M = d * inv(B)} has entries in R.
LowTriBddDenomInv:(M,R) -> M
++ LowTriBddDenomInv(B, d) returns M, where
++ B is a non-singular lower triangular matrix and d is an
++ element of R such that \( M = d \cdot \text{inv}(B) \) has entries in R.

Implementation => add

```
UpTriBddDenomInv(A, denom) ==
    AI := zero(nrows A, nrows A)$M
    offset := minColIndex AI - minRowIndex AI
    for i in minRowIndex AI .. maxRowIndex AI
        for j in minColIndex AI .. maxColIndex AI repeat
            qsetelt!(AI,i,j,(denom exquo qelt(A,i,j))::R)
    for i in minRowIndex AI .. maxRowIndex AI repeat
        for j in offset + i + 1 .. maxColIndex AI repeat
            qsetelt!(AI,i,j, - (((+\prod[k \in i+offset..(j-1)]
                qelt(A,k-offset,j) * qelt(A,k,i))
                for k in i-offset..(j-1)))
                exquo qelt(A, j-offset, j))::R))
    AI

LowTriBddDenomInv(A, denom) ==
    AI := zero(nrows A, nrows A)$M
    offset := minColIndex AI - minRowIndex AI
    for i in minRowIndex AI .. maxRowIndex AI
        for j in minColIndex AI .. maxColIndex AI repeat
            qsetelt!(AI,i,j,(denom exquo qelt(A,i,j))::R)
    for i in minRowIndex AI .. maxRowIndex AI repeat
        for j in i - offset + 1 .. maxRowIndex AI repeat
            qsetelt!(AI,j,i, - (((+\prod[k \in i-offset..(j-1)]
                qelt(A,j,k+offset) * qelt(AI,k,i))
                for k in i-offset..(j-1)))
                exquo qelt(A, j, j+offset))::R))
    AI
```
package TRIGMNIP TrigonometricManipulations

--- TrigonometricManipulations.input ---

)set break resume
)sys rm -f TrigonometricManipulations.output
)spool TrigonometricManipulations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TrigonometricManipulations
--E 1

)spool
)lisp (bye)

-----

--- TrigonometricManipulations.help ---

====================================================================
TrigonometricManipulations examples
====================================================================

TrigonometricManipulations provides transformations from trigonometric
functions to complex exponentials and logarithms, and back.

See Also:
o )show TrigonometricManipulations

-----
TrigonometricManipulations (TRIGMNIP)

Exports:
complexElementary complexForm complexNormalize imag real real? trigs

— package TRIGMNIP TrigonometricManipulations —

)abbrev package TRIGMNIP TrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 14 February 1994
++ Description:
++ \spadtype{TrigonometricManipulations} provides transformations from
++ trigonometric functions to complex exponentials and logarithms, and back.

TrigonometricManipulations(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)
  Z ==> Integer
  SY ==> Symbol
  K ==> Kernel F
  FG ==> Expression Complex R

Exports == with
  complexNormalize: F -> F
  ++ complexNormalize(f) rewrites \spad{f} using the least possible number
  ++ of complex independent kernels.
  complexNormalize: (F, SY) -> F
  ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
  ++ number of complex independent kernels involving \spad{x}.
  complexElementary: F -> F
  ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
  ++ complex transcendental elementary functions: \spad{log, exp}. 

complexElementary: (F, SY) -> F
  ++ complexElementary(f, x) rewrites the kernels of \(\text{spad\{f\}}\) involving
  ++ \(\text{spad\{x\}}\) in terms of the 2 fundamental complex
  ++ transcendental elementary functions: \(\text{spad\{log, exp\}}\).
trigs : F -> F
  ++ trigs(f) rewrites all the complex logs and exponentials
  ++ appearing in \(\text{spad\{f\}}\) in terms of trigonometric functions.
real : F -> F
  ++ real(f) returns the real part of \(\text{spad\{f\}}\) where \(\text{spad\{f\}}\) is a complex
  ++ function.
imag : F -> F
  ++ imag(f) returns the imaginary part of \(\text{spad\{f\}}\) where \(\text{spad\{f\}}\)
  ++ is a complex function.
real? : F -> Boolean
  ++ real?(f) returns \(\text{spad\{true\}}\) if \(\text{spad\{f = real f\}}\).
complexForm: F -> Complex F
  ++ complexForm(f) returns \(\text{spad\{[real f, imag f]\}}\).

Implementation ==> add
import ElementaryFunctionSign(R, F)
import InnerTrigonometricManipulations(R,F,FG)
import ElementaryFunctionStructurePackage(R, F)
import ElementaryFunctionStructurePackage(Complex R, FG)
s1 := sqrt(-1::F)
ipi := pi()$F * s1
K2KG : K -> Kernel FG
kcomplex : K -> Union(F, "failed")
locexplogs : F -> FG
localexplogs : (F, F, List SY) -> FG
complexKernels: F -> Record(ker: List K, val: List F)

K2KG k == retract(tan F2FG first argument k)@Kernel(FG)
real? f == empty?(complexKernels(f).ker)
real f == real complexForm f
imag f == imag complexForm f

complexKernels f ==
  lk:List(K) := empty()
  lv:List(F) := empty()
  for k in tower f repeat
    if (u := kcomplex k) case F then
      lk := concat(k, lk)
      lv := concat(u::F, lv)
  [lk, lv]

complexKernels f if it is certain that k is not a real kernel and k = f,
kcomplex k ==
  op := operator k
is?(k, "nthRoot"::SY) =>
  arg := argument k
  even?(retract(n := second arg)@Z) and ((u := sign(first arg)) case Z)
  and (u::Z < 0) => op(s1, n / 2::F) * op(- first arg, n)
"failed"

is?(k, "log"::SY) and ((u := sign(a := first argument k)) case Z)
  and (u::Z < 0) => op(- a) + ipi
"failed"

complexForm f ==
  empty?((l := complexKernels f).ker) => complex(f, 0)
  explogs2trigs locexplogs eval(f, l.ker, l.val)

locexplogs f ==
  any?(x +-> has?(x, "rtrig"),
    operators(g := realElementary f)$List(BasicOperator) =>
    locexplogs(f, g, variables g)
  )
  F2FG g

complexNormalize(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
    [k for k in tower(g := realElementary(f, x))
     | member?(x, variables(k::F))]$List(K)$List(K) =>
    FG2F(rischNormalize(locexplogs(f, g, [x]), x).func)
  )
  rischNormalize(g, x).func

complexElementary(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
    [k for k in tower(g := realElementary(f, x))
     | member?(x, variables(k::F))]$List(K)$List(K) =>
    FG2F locexplogs(f, g, [x])
  )
  g

complexElementary f ==
  any?(x +-> has?(x, "rtrig"),
    operators(g := realElementary f)$List(BasicOperator) =>
    FG2F locexplogs(f, g, variables g)
  )
  g
localexplogs(f, g, lx) ==
    trigs2explogs(F2FG g, [K2KG k for k in tower f
    | is?(k, "tan"::SY) or is?(k, "cot"::SY)], lx)

trigs f ==
    real? f => f
    g := explogs2trigs F2FG f
    real g + s1 * imag g

package TUBETOOL TubePlotTools
Tools for constructing tubes around 3-dimensional parametric curves.

See Also:
o )show TubePlotTools
++ point(x1,x2,x3,c) creates and returns a point from the three
++ specified coordinates \spad{x1}, \spad{x2}, \spad{x3}, and also
++ a fourth coordinate, c, which is generally used to specify the
++ color of the point.

"*" : (SF,Pt) -> Pt
++ s * p returns a point whose coordinates are the scalar multiple
++ of the point p by the scalar s, preserving the color, or fourth
++ coordinate, of p.

"+" : (Pt,Pt) -> Pt
++ p + q computes and returns a point whose coordinates are the sums
++ of the coordinates of the two points \spad{p} and \spad{q}, using
++ the color, or fourth coordinate, of the first point \spad{p}
++ as the color also of the point \spad{q}.

"-" : (Pt,Pt) -> Pt
++ p - q computes and returns a point whose coordinates are the
++ differences of the coordinates of two points \spad{p} and \spad{q},
++ using the color, or fourth coordinate, of the first point \spad{p}
++ as the color also of the point \spad{q}.

dot : (Pt,Pt) -> SF
++ dot(p,q) computes the dot product of the two points \spad{p}
++ and \spad{q} using only the first three coordinates, and returns
++ the resulting \spadtype{DoubleFloat}.

cross : (Pt,Pt) -> Pt
++ cross(p,q) computes the cross product of the two points \spad{p}
++ and \spad{q} using only the first three coordinates, and keeping
++ the color of the first point p. The result is returned as a point.

unitVector: Pt -> Pt
++ unitVector(p) creates the unit vector of the point p and returns
++ the result as a point. Note that \spad{unitVector(p) = p/|p|}.

cosSinInfo: I -> L L SF
++ cosSinInfo(n) returns the list of lists of values for n, in the form
++ \spad{[[cos(n-1) a,sin(n-1) a],...,[cos 2 a,sin 2 a],[cos a,sin a]]}
++ where \spad{a = 2 \pi/n}. Note that n should be greater than 2.

loopPoints: (Pt,Pt,Pt,SF,L L SF) -> L Pt
++ loopPoints(p,n,b,r,lls) creates and returns a list of points
++ which form the loop with radius r, around the center point
++ indicated by the point p, with the principal normal vector of
++ the space curve at point p given by the point(vector) n, and the
++ binormal vector given by the point(vector) b, and a list of lists,
++ lls, which is the \spadfun{cosSinInfo} of the number of points
++ defining the loop.

Implementation ==> add
import PointPackage(SF)

point(x,y,z,c) == point(l : L SF := [x,y,z,c])

getColor: Pt -> SF
getColor pt == (maxIndex pt > 3 => color pt; 0)
getColor2: (Pt,Pt) -> SF
getColor2(p0,p1) ==
  maxIndex p0 > 3 => color p0
  maxIndex p1 > 3 => color p1
  0

a * p ==
  l : L SF := [a * xCoord p,a * yCoord p,a * zCoord p,getColor p]
  point l

p0 + p1 ==
  l : L SF := [xCoord p0 + xCoord p1,yCoord p0 + yCoord p1,_,
    zCoord p0 + zCoord p1,getColor2(p0,p1)]
  point l

p0 - p1 ==
  l : L SF := [xCoord p0 - xCoord p1,yCoord p0 - yCoord p1,_,
    zCoord p0 - zCoord p1,getColor2(p0,p1)]
  point l

dot(p0,p1) ==
  (xCoord p0 * xCoord p1) + (yCoord p0 * yCoord p1) +
  (zCoord p0 * zCoord p1)

cross(p0,p1) ==
  x0 := xCoord p0; y0 := yCoord p0; z0 := zCoord p0;
  x1 := xCoord p1; y1 := yCoord p1; z1 := zCoord p1;
  l : L SF := [y0 * z1 - y1 * z0,z0 * x1 - z1 * x0,_,
    x0 * y1 - x1 * y0,getColor2(p0,p1)]
  point l

unitVector p == (inv sqrt dot(p,p)) * p

cosSinInfo n ==
  ans : L L SF := nil()
  theta : SF := 2 * pi()/n
  for i in 1..(n-1) repeat --!! make more efficient
    angle := i * theta
    ans := concat([cos angle,sin angle],ans)
  ans

loopPoints(ctr,pNorm,bNorm,rad,cosSin) ==
  ans : L Pt := nil()
  while not null cosSin repeat
    cossin := first cosSin; cos := first cossin; sin := second cossin
    ans := cons(ctr + rad * (cos * pNorm + sin * bNorm),ans)
    cosSin := rest cosSin
  pt := ctr + rad * pNorm
  concat(pt,concat(ans,pt))
package CLIP TwoDimensionalPlotClipping

---

TwoDimensionalPlotClipping examples

Automatic clipping for 2-dimensional plots. The purpose of this package is to provide reasonable plots of functions with singularities.

See Also:
- )show TwoDimensionalPlotClipping

---
TwoDimensionalPlotClipping (CLIP)

Exports:
clip  clipParametric  clipWithRanges

--- package CLIP TwoDimensionalPlotClipping ---

TwoDimensionalPlotClipping(): Exports == Implementation where
B ==> Boolean
L ==> List
SEG ==> Segment
RN ==> Fraction Integer
SF ==> DoubleFloat
Pt ==> Point DoubleFloat
PLOT ==> Plot
CLIPPED ==> Record(brans: L L Pt,xValues: SEG SF,yValues: SEG SF)

Exports ==> with
clip: PLOT -> CLIPPED
++ clip(p) performs two-dimensional clipping on a plot, p, from
++ the domain \texttt{Plot} for the graph of one variable,
++ \texttt{spad(y = f(x))}; the default parameters \texttt{1/4} for the fraction
++ and \texttt{5/1} for the scale are used in the \texttt{spadfun\{clip\}} function.
clip: (PLOT,RN,RN) -> CLIPPED
++ clip(p,frac,sc) performs two-dimensional clipping on a plot, p,
CHAPTER 21. CHAPTER T

++ from the domain \spadtype{Plot} for the graph of one variable
++ \spad{y = f(x)}; the fraction parameter is specified by \spad{frac}
++ and the scale parameter is specified by \spad{sc} for use in the
++ \spadfun{clip} function.

\textbf{clipParametric: PLOT} \to \textbf{CLIPPED}
++ clipParametric(p) performs two-dimensional clipping on a plot,
++ p, from the domain \spadtype{Plot} for the parametric curve
++ \spad{x = f(t)}, \spad{y = g(t)}; the default parameters \spad{1/2}
++ for the fraction and \spad{5/1} for the scale are used in the
++ \spad{fakeAxiomFun{iClipParametric}} subroutine, which is called by this
++ function.

\textbf{clipParametric: (PLOT,RN,RN)} \to \textbf{CLIPPED}
++ clipParametric(p,frac,sc) performs two-dimensional clipping on a
++ plot, p, from the domain \spadtype{Plot} for the parametric curve
++ \spad{x = f(t)}, \spad{y = g(t)}; the fraction parameter is
++ specified by \spad{frac} and the scale parameter is specified
++ by \spad{sc} for use in the \spad{fakeAxiomFun{iClipParametric}} subroutine,
++ which is called by this function.

\textbf{clipWithRanges: (L L Pt, SF, SF, SF, SF)} \to \textbf{CLIPPED}
++ clipWithRanges(pointLists, xMin, xMax, yMin, yMax) performs clipping
++ on a list of lists of points, \spad{pointLists}. Clipping is
++ done within the specified ranges of \spad{xMin}, \spad{xMax} and
++ \spad{yMin}, \spad{yMax}. This function is used internally by
++ the \spad{fakeAxiomFun{iClipParametric}} subroutine in this package.

\textbf{clip: L Pt} \to \textbf{CLIPPED}
++ clip(l) performs two-dimensional clipping on a curve l, which is
++ a list of points; the default parameters \spad{1/2} for the
++ fraction and \spad{5/1} for the scale are used in the
++ \spad{fakeAxiomFun{iClipParametric}} subroutine, which is called by this
++ function.

\textbf{clip: L L Pt} \to \textbf{CLIPPED}
++ clip(ll) performs two-dimensional clipping on a list of lists
++ of points, \spad{ll}; the default parameters \spad{1/2} for
++ the fraction and \spad{5/1} for the scale are used in the
++ \spad{fakeAxiomFun{iClipParametric}} subroutine, which is called by this
++ function.

\textbf{Implementation} \to add

import PointPackage(DoubleFloat)
import ListFunctions2(Point DoubleFloat, DoubleFloat)

point: (SF, SF) \to Pt
intersectWithHorizLine: (SF, SF, SF, SF, SF) \to Pt
intersectWithVertLine: (SF, SF, SF, SF, SF) \to Pt
intersectWithBdry: (SF, SF, SF, Pt, Pt) \to Pt
discardAndSplit: (L Pt, Pt) \to B, SF, SF, SF, SF) \to L L Pt
norm: Pt \to SF
iClipParametric: (L L Pt, RN, RN) \to CLIPPED
findPt: L L Pt \to Union(Pt, "failed")
Fnan?: SF \to Boolean
\begin{verbatim}
Pnan?:Pt ->Boolean
Fnan? x == x"=x
Pnan? p == any?(Fnan?,p)

iClipParametric(pointLists,fraction,scale) ==
  -- error checks and special cases
  (fraction < 0) or (fraction > 1) =>
  error "clipDraw: fraction should be between 0 and 1"
empty? pointLists => [nil(),segment(0,0),segment(0,0)]
  -- put all points together , sort them according to norm
sortedList := sort((x:Pt,y:Pt):Boolean +-> norm(x) < norm(y),
                  select((z:Pt):Boolean +-> not Pnan? z,concat pointLists))
empty? sortedList => [nil(),segment(0,0),segment(0,0)]
n := # sortedList
num := numer fraction
den := denom fraction
clipNum := (n * num) quo den
lastN := n - 1 - clipNum
firstPt := first sortedList
xMin : SF := xCoord firstPt
xMax : SF := xCoord firstPt
yMin : SF := yCoord firstPt
yMax : SF := yCoord firstPt
  -- calculate min/max for the first (1-fraction)*N points
  -- this contracts the range
  -- this unnecessarily clips monotonic functions (step-function, x"(high power),etc.)
for k in 0..lastN for pt in rest sortedList repeat
  xMin := min(xMin,xCoord pt)
  xMax := max(xMax,xCoord pt)
  yMin := min(yMin,yCoord pt)
  yMax := max(yMax,yCoord pt)
  xDiff := xMax - xMin; yDiff := yMax - yMin
  xDiff = 0 =>
    yDiff = 0 =>
      [pointLists,segment(xMin-1,xMax+1),segment(yMin-1,yMax+1)]
      [pointLists,segment(xMin-1,xMax+1),segment(yMin-1,yMax+1)]
yDiff = 0 =>
  [pointLists,segment(xMin,xMax),segment(yMin-1,yMax+1)]
numm := numer scale; denn := denom scale
  -- now expand the range by scale
  xMin := xMin + (numm :: SF) * xDiff / (denn :: SF)
  xMax := xMax + (numm :: SF) * xDiff / (denn :: SF)
  yMin := yMin + (numm :: SF) * yDiff / (denn :: SF)
  yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
  -- clip with the calculated range
  newclip:=clipWithRanges(pointLists,xMin,xMax,yMin,yMax)
  -- if we split the lists use the new clip
  # (newclip.brans) > # pointLists => newclip
  -- calculate extents
\end{verbatim}
xs : L SF := map (xCoord, sortedList)
y : L SF := map (yCoord, sortedList)
xMin : SF := reduce (min, xs)
yMin : SF := reduce (min, ys)
xMax : SF := reduce (max, xs)
yMax : SF := reduce (max, ys)
xSeg : SEG SF := xMin..xMax
ySeg : SEG SF := yMin..yMax
-- return original
[pointLists, xSeg, ySeg]@CLIPPED

point(xx, yy) == point(l : L SF := [xx, yy])

intersectWithHorizLine(x1, y1, x2, y2, yy) ==
  x1 = x2 => point(x1, yy)
  point(x1 + (x2 - x1)*(yy - y1)/(y2 - y1), yy)

intersectWithVertLine(x1, y1, x2, y2, xx) ==
  y1 = y2 => point(xx, y1)
  point(xx, y1 + (y2 - y1)*(xx - x1)/(x2 - x1))

intersectWithBdry(xMin, xMax, yMin, yMax, pt1, pt2) ==
  -- pt1 is in rectangle, pt2 is not
  x1 := xCoord pt1; y1 := yCoord pt1
  x2 := xCoord pt2; y2 := yCoord pt2
  if y2 > yMax then
    pt2 := intersectWithHorizLine(x1, y1, x2, y2, yMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if y2 < yMin then
    pt2 := intersectWithHorizLine(x1, y1, x2, y2, yMin)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 > xMax then
    pt2 := intersectWithVertLine(x1, y1, x2, y2, xMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 < xMin then
    pt2 := intersectWithVertLine(x1, y1, x2, y2, xMin)
    pt2

discardAndSplit(pointList, pred, xMin, xMax, yMin, yMax) ==
  ans : L L Pt := nil()
  list : L Pt := nil()
  lastPt? : B := false
  lastPt : Pt := point(0, 0)
  while not empty? pointList repeat
    pt := first pointList
    pointList := rest pointList
    pred(pt) =>

if (empty? list) and lastPt? then
  bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,pt,lastPt)
  -- print bracket [ coerce bdryPt ,coerce pt ]
  --list := cons(bdryPt,list)
list := cons(pt,list)
if not empty? list then
  bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,first list,pt)
  -- print bracket [ coerce bdryPt,coerce first list]
  --list := cons(bdryPt,list)
ans := cons( list,ans)
lastPt := pt
lastPt? := true
list := nil()
empty? list => ans
reverse_! cons(reverse_! list,ans)

clip(plot,fraction,scale) ==
-- sayBrightly("clip: ":OutputForm$List(OutputForm))$Lisp
(fraction < 0) or (fraction > 1/2) =>
  error "clipDraw: fraction should be between 0 and 1/2"
xVals := xRange plot
empty?(pointLists := listBranches plot) =>
  [nil(),xVals,segment(0,0)]
more?(pointLists := listBranches plot,1) =>
  error "clipDraw: plot has more than one branch"
empty?(pointList := first pointLists) =>
  [nil(),xVals,segment(0,0)]
sortedList := sort((x,y)+->yCoord(x) < yCoord(y),pointList)
n := # sortedList; num := numer fraction; den := denom fraction
clipNum := (n * num) quo den
-- throw out points with large and small y-coordinates
yMin := yCoord(sortedList.clipNum)
yMax := yCoord(sortedList.(n - 1 - clipNum))
if Fnan? yMin then yMin := 0
if Fnan? yMax then yMax := 0
(yDiff := yMax - yMin) = 0 =>
  [pointLists,xRange plot,segment(yMin - 1,yMax + 1)]
numm := numer scale; denn := denom scale
xmin := lo xVals; xmax := hi xVals
yMin := yMin - (numm :: SF) * yDiff / (denn :: SF)
yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
lists := discardAndSplit(pointList,_,
  x +-> (yCoord(x) < yMax) and (yCoord(x) > yMin),
  xmin,xMax,yMin,yMax)
yMin := yCoord(sortedList.clipNum)
yMax := yCoord(sortedList.(n - 1 - clipNum))
if Fnan? yMin then yMin := 0
if Fnan? yMax then yMax := 0
for list in lists repeat
  for pt in list repeat
if not Fnan?(yCoord pt) then
  yMin := min(yMin,yCoord pt)
yMax := max(yMax,yCoord pt)

lists,xVals,segment(yMin,yMax)]

clip(plot:PLOT) == clip(plot,1/4,5/1)

norm(pt) ==
x := xCoord(pt); y := yCoord(pt)
if Fnan? x then
  if Fnan? y then
    r:SF := 0
  else
    r:SF := y**2
else
  if Fnan? y then
    r:SF := x**2
  else
    r:SF := x**2 + y**2
r

findPt lists ==
  for list in lists repeat
    not empty? list =>
      for p in list repeat
        not Pnan? p => return p
    "failed"

clipWithRanges(pointLists,xMin,xMax,yMin,yMax) ==
  lists : L L Pt := nil()
  for pointList in pointLists repeat
    lists := concat(lists,discardAndSplit(pointList,
      (x:Pt):Boolean +-> (xCoord(x) <= xMax) and (xCoord(x) >= xMin) and
      (yCoord(x) <= yMax) and (yCoord(x) >= yMin),
      xMin,xMax,yMin,yMax))
    (pt := findPt lists) case "failed" =>
      [nil(),segment(0,0),segment(0,0)]
  firstPt := pt :: Pt
  xMin : SF := xCoord firstPt; xMax : SF := xCoord firstPt
  yMin : SF := yCoord firstPt; yMax : SF := yCoord firstPt
  for list in lists repeat
    for pt in list repeat
      if not Pnan? pt then
        xMin := min(xMin,xCoord pt)
xMax := max(xMax,xCoord pt)
yMin := min(yMin,yCoord pt)
yMax := max(yMax,yCoord pt)
  [lists,segment(xMin,xMax),segment(yMin,yMax)]

clipParametric(plot,fraction,scale) ==
iClipParametric(listBranches plot,fraction,scale)
clipParametric plot == clipParametric(plot,1/2,5/1)
clip(l: L Pt) == iClipParametric(list l,1/2,5/1)
clip(l: L L Pt) == iClipParametric(l,1/2,5/1)

---

package TWOFACT TwoFactorize

--- TwoFactorize.input ---

)set break resume
)sys rm -f TwoFactorize.output
)spool TwoFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TwoFactorize
--E 1

)spool
)lisp (bye)

---

--- TwoFactorize.help ---

====================================================================
TwoFactorize examples
====================================================================
A basic package for the factorization of bivariate polynomials over a finite field. The functions here represent the base step for the multivariate factorizer.

See Also:
  o )show TwoFactorize

---

TwoFactorize (TWOFACT)

Exports:
  generalSqFr  generalTwoFactor  twoFactor

— package TWOFACT TwoFactorize —

)abbrev package TWOFACT TwoFactorize
++ Authors : P.Gianni, J.H.Davenport
++ Date Created : May 1990
++ Date Last Updated: March 1992
++ Description:
  ++ A basic package for the factorization of bivariate polynomials
  ++ over a finite field.
  ++ The functions here represent the base step for the multivariate factorizer.

TwoFactorize(F) : C == T
where
  F       :  FiniteFieldCategory
  SUP     =>  SparseUnivariatePolynomial
  R       =>  SUP F
  P       =>  SUP R
  UPCF2   =>  UnivariatePolynomialCategoryFunctions2
C == with
  generalTwoFactor : (P) -> Factored P
    ++ generalTwoFactor(p) returns the factorisation of polynomial p,
    ++ a sparse univariate polynomial (sup) over a sup over F.
  generalSqFr : (P) -> Factored P
    ++ generalSqFr(p) returns the square-free factorisation of polynomial p,
    ++ a sparse univariate polynomial (sup) over a sup over F.
  twoFactor : (P,Integer) -> Factored P
    ++ twoFactor(p,n) returns the factorisation of polynomial p,
    ++ a sparse univariate polynomial (sup) over a sup over F.
    ++ Also, p is assumed primitive and square-free and n is the degree of the inner variable of p (maximum of the degrees of the coefficients of p).

T == add
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  import CommuteUnivariatePolynomialCategory(F,R,P)

---- Local Functions ----
computeDegree : (P,Integer,Integer) -> PI
exchangeVars : P -> P
exchangeVarTerm: (R, NNI) -> P
pthRoot : (R, NNI, NNI) -> R

-- compute the degree of the extension to reduce the polynomial to a univariate one
computeDegree(m : P,mx:Integer,q:Integer): PI ==
  my:=degree m
  n1:Integer:=length(10*mx*my)
  n2:Integer:=length(q)-1
  n:=(n1 quo n2)+1
  n::PI
-- n=1 => 1$PositiveInteger
-- (nextPrime(max(n,min(mx,my)))$IntegerPrimesPackage(Integer))::PI

exchangeVars(p : P) : P ==
  p = 0 => 0
  exchangeVarTerm(leadingCoefficient p, degree p) +
      exchangeVars(reductum p)

exchangeVarTerm(c:R, e:NNI) : P ==
  c = 0 => 0
  monomial(monomial(leadingCoefficient c, e)$R, degree c)$P +
      exchangeVarTerm(reductum c, e)

tmp:=divideExponents(map((x:F):F+->(x::F)**PthRootPow,poly),p)
tmp case "failed" => error "consistency error in TwoFactor"
tmp

fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

generalSqFr(m:P): Factored P ==
m = 0 => 0
degree m = 0 =>
l:=squareFree(leadingCoefficient m)
makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
cont := content m
m := (m exquo cont)::P
sqfrm := squareFree m
pfaclist : List FF := empty()
unitPart := unit sqfrm
for u in factorList sqfrm repeat
  u.flg = "nil" =>
    uexp:NNI:=(u.xpnt):NNI
    nfacs:=squareFree(exchangeVars u.fctr)
    for v in factorList nfacs repeat
      pfaclist:=cons([v.flg, exchangeVars v.fctr, v.xpnt*uexp],
                     pfaclist)
    unitPart := unit(nfacs)**uexp * unitPart
  pfaclist := cons(u,pfaclist)
cont ^= 1 =>
  sqp := squareFree cont
  contlist:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList sqp]
  pfaclist:= append(contlist, pfaclist)
  makeFR(unit(sqp)*unitPart, pfaclist)
makeFR(unitPart, pfaclist)

generalTwoFactor(m:P): Factored P ==
m = 0 => 0
degree m = 0 =>
l:=factor(leadingCoefficient m)$DistinctDegreeFactorize(F,R)
makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
ll:List FF
ll:=[[]
unitPart:P
cont:==content m
if degree(cont)>0 then
  m:=m exquo cont
  m case "failed" => error "content doesn’t divide"
m:=m1
  contfact:=factor(cont)$DistinctDegreeFactorize(F,R)
  unitPart:=(unit contfact)::P
  ll:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList contfact]
else
    unitPart:=cont::P
sqfrm:=squareFree m
for u in factors sqfrm repeat
    expo:=u.exponent
if expo < 0 then error "negative exponent in a factorisation"
expo:NonNegativeInteger:=expo::NonNegativeInteger
fac:=u.factor
degree fac = 1 => ll:=[["irred",fac,expon],:ll]
differentiate fac = 0 =>
    -- the polynomial is inseparable w.r.t. its main variable
    map(differentiate,fac) = 0 =>
p:=characteristic$F
PthRootPow:=(size$F exquo p)::NonNegativeInteger
m1:=(divideExponents(map(x+->pthRoot(x,p,PthRootPow),fac),p)
ml case "failed" => error "consistency error in TwoFactor"
res:=generalTwoFactor m1
unitPart:=unitPart*unit(res)**((p*expon)::NNI)
ll:=
    [:[[v.flg,v.fctr,expon*p*v.xpnt] for v in factorList res],:ll]

m2:=generalTwoFactor swap fac
unitPart:=unitPart*unit(m2)**(expon::NNI)
ll:=
    [:[[v.flg,swap v.fctr,expon*v.xpnt] for v in factorList m2],:ll]
ydeg="max"/[degree w for w in coefficients fac]
twoF:=twoFactor(fac,ydeg)
unitPart:=unitPart*unit(twoF)**expon
ll:=
    [:[[v.flg,v.fctr,expon*v.xpnt] for v in factorList twoF],
    :ll]
makeFR(unitPart,ll)

-- factorization of a primitive square-free bivariate polynomial --
twoFactor(m::P,F::Integer):Factored P ==
    -- choose the degree for the extension
n:PI:=computeDegree(m,F::Integer)$F
    -- extend the field
    -- find the substitution for x
look:Boolean:=true
dm:=degree m
try:=min(5,size()$F)
i:=0
lcm := leadingCoefficient m
umv : R
while look and i < try repeat
    vval := random()$R
    i:=i+1
    zero? elt(lcm, vval) => "next value"
    umv := map(x +-> elt(x,vval), m)$UPCF2(R, P, F, R)
degree(gcd(umv,differentiate umv))=0 => "next val"
n := 1
look := false
extField:=FiniteFieldExtension(F,n)
SUEx:=SUP extField
TP:=SparseUnivariatePolynomial SUEx
mm:=TP:=0
m1:=m
while m1^=0 repeat
    mm:=mm+monomial(map(coerce,leadingCoefficient m1)$UPCF2(F,R,extField,SUEx),degree m1)
m1:=reductum m1
lcmm := leadingCoefficient mm
val : extField
umex : SUEx
if not look then
    val := vval :: extField
    umex := map(coerce, umv)$UPCF2(F, R, extField, SUEx)
while look repeat
    val:=random()$extField
    i:=i+1
    elt(lcmm, val)=0 => "next value"
    umex := map(x +-> elt(x, val), mm)$UPCF2(SUEx, TP, extField, SUEx)
degree(gcd(umex,differentiate umex))^=0 => "next val"
look:=false
prime:SUEx:=monomial(1,1)-monomial(val,0)
fumex:=factor(umex)$DistinctDegreeFactorize(extField,SUEx)
lfact1:=factors fumex

#lfact1=1 => primeFactor(m,1)
lfact : List TP :=
    [map(coerce,lf.factor)$UPCF2(extField,SUEx,SUEx,TP)
     for if in lfact1]
import GeneralHenselPackage(SUEx,TP)
dx1:PI:=(dx+1)::PI
lfacth:=completeHensel(mm,lfact,prime,dx1)
lfactk: List P :=[]
Normp := NormRetractPackage(F, extField, SUEx, TP, n)

while not empty? lfacth repeat
    ff := first lfacth
    lfact := rest lfacth
    if (c:=leadingCoefficient leadingCoefficient ff) ^=1 then
        ff:=((inv c)::SUEx)* ff
    not ((ffu:= retractIfCan(ff)$Normp) case "failed") =>
        lfactk := cons(ffu::P, lfactk)
normfacs := normFactors(ff)$Normp
        lfact := [g for g in lfact | not member?(g, normfacs)]
    ffn := */normfacs
    lfactk:=cons(retractIfCan(ffn)$Normp :: P, lfactk)
*/[primeFactor(ff1,1) for ff1 in lfactk]

---

— TWOFACT.dotabb —

"TWOFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TWOFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"TWOFACT" -> "PFECAT"
"TWOFACT" -> "FAXF"

---
Chapter 22

Chapter U

package UNIFACT UnivariateFactorize

— UnivariateFactorize.input —

)set break resume
)sys rm -f UnivariateFactorize.output
)spool UnivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateFactorize
--E 1

)spool
)lisp (bye)

— UnivariateFactorize.help —

====================================================================
UnivariateFactorize examples
====================================================================

Package for the factorization of univariate polynomials with integer coefficients. The factorization is done by "lifting" (HENSEL) the factorization over a finite field.
See Also:
- )show UnivariateFactorize

**UnivariateFactorize (UNIFACT)**

Exports:
- factor
- factorSquareFree
- henselFact

--- package UNIFACT UnivariateFactorize ---

)abbrev package UNIFACT UnivariateFactorize
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: December 1993
++ Description:
++ Package for the factorization of univariate polynomials with integer
++ coefficients. The factorization is done by "lifting" (HENSEL) the
++ factorization over a finite field.

UnivariateFactorize(ZP) : public == private where
  Z ==> Integer
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  SUPZ ==> SparseUnivariatePolynomial Z

  ZP : UnivariatePolynomialCategory Z

  FR ==> Factored ZP
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FFE ==> Record(flgl:fUnion, fctr:ZP, xpnt:Z)
  ParFact ==> Record(irr: ZP,pow: Z)
FinalFact == Record(contp: Z, factors: List(ParFact))

Public == with
factor : ZP -> FR
++ factor(m) returns the factorization of m
factorSquareFree : ZP -> FR
++ factorSquareFree(m) returns the factorization of m square free polynomial
henselFact : (ZP,Boolean) -> FinalFact
++ henselFact(m,flag) returns the factorization of m,
++ FinalFact is a Record s.t. FinalFact.contp=content m,
++ FinalFact.factors=List of irreducible factors
++ of m with exponent , if flag =true the polynomial is
++ assumed square free.

Private == add

--- local functions ---

henselFact : ZP -> List(ZP)
quadratic : ZP -> List(ZP)
remp : (Z, PI) -> Z
negShiftz : (Z, PI) -> Z
negShiftp : (ZP, PI) -> ZP
bound : ZP -> PI
choose : ZP -> FirstStep
eisenstein : ZP -> Boolean
isPowerOf2 : Z -> Boolean
subMinusX : SUPZ -> ZP
sqroot : Z -> Z

--- declarations ---

CYC ==> CyclotomicPolynomialPackage()
DDRecord ==> Record(factor: ZP, degree: Z)
DDList ==> List DDRecord
FirstStep ==> Record(prime: PI, factors: DDList)
ContPrim ==> Record(cont: Z, prim: ZP)

import GeneralHenselPackage(Z,ZP)
import ModularDistinctDegreeFactorizer ZP

factor(m: ZP) ==
  flist := henselFact(m,false)
  ctp:=unitNormal flist.contp
  makeFR((ctp.unit)::ZP, cons(["nil", ctp.canonical::ZP, 1$Z]$FFE,
                            [["prime", u.irr, u.pow]$FFE for u in flist.factors]))

factorSquareFree(m: ZP) ==
  flist := henselFact(m,true)
--- Integer square root: returns 0 if t is non-positive
sqroot(t: Z): Z ==
  t <= 0 => 0
  s:Integer:=t::Integer
  s:=approxSqrt(s)$IntegerRoots(Integer)
  t:=s::Z
  t

--- Eisenstein criterion: returns true if polynomial is
--- irreducible. Result of false in inconclusive.
eisenstein(m : ZP): Boolean ==
  -- calculate the content of the terms after the first
  c := content reductum m
  c = 0 => false
  c = 1 => false
  -- factor the content
  -- if there is a prime in the factorization that does not divide
  -- the leading term and appears to multiplicity 1, and the square
  -- of this does not divide the last coef, return true.
  -- Otherwise return false.
  lead := leadingCoefficient m
  trail := lead
  m := reductum m
  while m ^= 0 repeat
    trail := leadingCoefficient m
    m:= reductum m
    fc := factor(c) :: Factored(Z)
    for r in factors fc repeat
      if (r.exponent = 1) and (0 ^= (lead rem r.factor)) and
        (0 ^= (trail rem (r.factor ** 2))) then return true
    false

negShiftz(n: Z,Modulus:PI): Z ==
  if n < 0 then n := n+Modulus
  n > (Modulus quo 2) => n-Modulus
  n

negShiftp(pp: ZP,Modulus:PI): ZP ==
  map(x +-> negShiftz(x,Modulus),pp)

--- Choose the bound for the coefficients of factors
bound(m: ZP):PI ==
  nm,nmq2,lcm,bin0,bin1:NNI
  cbound,j : PI
  k:NNI
lcm := abs(leadingCoefficient m)::NNI
nm := (degree m)::NNI
nmq2 := nm quo 2
norm: Z := sqrt(+[coefficient(m,k)**2 for k in 0..nm])
if nmq2^=1 then nm := (nmq2-1):NNI
else nm := nmq2
bin0 := nm
cbound := (bin0*norm+lcm)::PI
for i in 2..(nm-1)::NNI repeat
  bin1 := bin0
  bin0 := (bin0*(nm+1-i):NNI) quo i
  j := (bin0*norm+bin1*lcm)::PI
  if cbound<j then cbound := j
(2*cbound*lcm)::PI -- adjusted by lcm to prepare for exquo in ghensel

remp(t: Z,q:PI): Z == ((t := t rem q)<0 => t+q ;t)

numFactors(ddlist:DDList): Z ==
  ans: Z := 0
  for dd in ddlist repeat
    (d := degree(dd.factor)) = 0 => nil
    ans := ans + ((d pretend Z) exquo dd.degree):: Z
  ans
  -- select the prime,try up to 4 primes,
  -- choose the one yielding the fewest factors, but stopping if
  -- fewer than 9 factors
choose(m: ZP):FirstStep ==
  qSave:PI := 1
  ddSave:DDList := []
  numberOfFactors: Z := 0
  lcm := leadingCoefficient m
  k: Z := 1
  ddRep := 5
  disc:ZP:=0
  q:PI:=2
  while k<ddRep repeat
    -- q must be a new prime number at each iteration
    q:=nextPrime(q)$IntegerPrimesPackage(Z) pretend PI
    (rr:=lcm rem q) = 0$Z => "next prime"
    disc:=gcd(m,differentiate m,q)
    (degree disc)^=0 => "next prime"
    k := k+1
    newdd := ddFact(m,q)
    ((n := numFactors(newdd)) < 9) =>
      ddSave := newdd
      qSave := q
      k := 5
    (numberOfFactors = 0) or (n < numberOfFactors) =>
      ddSave := newdd

qSave := q
numberOffFactors := n
[qSave,ddSave]$FirstStep

-- Find the factors of m, primitive, square-free, with lc positive
-- and mindeg m = 0
henselfact1(m: ZP):List(ZP) ==
  zero? degree m =>
  (m = 1) => []
(m = 1) => []

selected := choose(m)
(numFactors(selected.factors) = 1$Z) => [m]
q := selected.prime
f1 := separateFactors(selected.factors, q)
-- choose the bound
cbound := bound(m)
completeHensel(m, f1, q, cbound)

-- check for possible degree reduction
-- could use polynomial decomposition?
henselfact(m: ZP):List ZP ==
  deggcd := degree m
  mm := m
  while not zero? mm repeat (deggcd := gcd(ddegcd, degree mm); mm := reductum mm)
  deggcd > 1 and deggcd < degree m =>
    faclist := henselfact1(divideExponents(m, deggcd)::ZP)
    "append"/[henselfact1 multiplyExponents(mm, deggcd) for mm in faclist]
  henselfact1 m

quadratic(m: ZP):List(ZP) ==
d, d2: Z
  d := coefficient(m, 1)**2 - 4*coefficient(m, 0)*coefficient(m, 2)
  d2 := sqroot(d)
  (d-d2**2)^=0 => [m]
  alpha: Z := coefficient(m, 1)+d2
  beta: Z := 2*coefficient(m, 2)
  d := gcd(alpha, beta)
  if d ^= 1 then
    alpha := alpha quo d
    beta := beta quo d
    m0: ZP := monomial(beta, 1)+monomial(alpha, 0)
    cons(m0, [(m exquo m0):: ZP])

isPowerOf2(n : Z): Boolean ==
n = 1 => true
qr : Record(quotient: Z, remainder: Z) := divide(n, 2)
qr.remainder = 1 => false
isPowerOf2 qr.quotient
subMinusX(supPol : SUPZ): ZP ==
  minusX : SUPZ := monomial(-1,1)$SUPZ
  (elt(supPol,minusX)$SUPZ) : ZP

-- Factorize the polynomial m, test=true if m is known to be
-- square-free, false otherwise.
-- FinalFact.contp=content m, FinalFact.factors=List of irreducible
-- factors with exponent .
henselFact(m: ZP,test:Boolean):FinalFact ==
  factorlist : List(ParFact) := []
  c : Z

  -- make m primitive
  c := content m
  m := (m exquo c)::ZP

  -- make the lc m positive
  if leadingCoefficient m < 0 then
    c := -c
    m := -m

  -- is x**d factor of m?
  if (d := minimumDegree m) >0 then
    m := (monicDivide(m,monomial(1,d))).quotient
    factorlist := [[monomial(1,1),d]$ParFact]
  d := degree m

  -- is m constant?
  d=0 => [c,factorlist]$FinalFact

  -- is m linear?
  d=1 => [c,cons([m,1]$ParFact,factorlist)]$FinalFact

  -- does m satisfy Eisenstein's criterion?
  eisenstein m => [c,cons([m,1]$ParFact,factorlist)]$FinalFact

  lcPol : ZP := leadingCoefficient(m) :: ZP

  -- is m cyclotomic (x**n - 1)?
  -lcPol = reductum(m) => -- if true, both will = 1
    for fac in (cyclotomicDecomposition(degree m)$CYC : List ZP) repeat
      factorlist := cons([fac,1]$ParFact,factorlist)
    [c,factorlist]$FinalFact

  -- is m odd cyclotomic (x**(2*n+1) + 1)?
  odd?(d) and (lcPol = reductum(m)) =>
    for sfac in cyclotomicDecomposition(degree m)$CYC repeat
      fac:=subMinusX sfac
if leadingCoefficient fac < 0 then fac := -fac
factorlist := cons([fac,1]$ParFact,factorlist)
[c,factorlist]$$FinalFact

-- is the poly of the form x**n + 1 with n a power of 2?
-- if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(m)) =>
factorlist := cons([m,1]$ParFact,factorlist)
[c,factorlist]$FinalFact

-- is m quadratic?
d=2 =>
lfq:List(ZP) := quadratic m
#lfq=1 => [c,cons([lfq.first,1]$ParFact,factorlist)]$FinalFact
(lf0,lf1) := (lfq.first,second lfq)
if lf0=lf1 then factorlist := cons([lf0,2]$ParFact,factorlist)
else factorlist := append([[v,1]$ParFact for v in lfq],factorlist)
[c,factorlist]$FinalFact

-- m is square-free
test =>
fln := henselfact(m)
[c,append(factorlist,[[pf,1]$ParFact for pf in fln]])$FinalFact

-- find the square-free decomposition of m
irrFact := squareFree(m)
llf := factors irrFact

-- factorize the square-free primitive terms
for l1 in llf repeat
d1 := l1.exponent
pol := l1.factor
degree pol=1 => factorlist := cons([pol,d1]$ParFact,factorlist)
degree pol=2 =>
fln := quadratic(pol)
factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
fln := henselfact(pol)
factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
[c,factorlist]$FinalFact

— UNIFACT.dotabb —

"UNIFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UNIFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UNIFACT" -> "PFECAT"
package UFPS1 UnivariateFormalPowerSeriesFunctions

---

--- UnivariateFormalPowerSeriesFunctions.input ---

)set break resume
)sys rm -f UnivariateFormalPowerSeriesFunctions.output
)spool UnivariateFormalPowerSeriesFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateFormalPowerSeriesFunctions
--E 1

)spool
)lisp (bye)

---

--- UnivariateFormalPowerSeriesFunctions.help ---

====================================================================
UnivariateFormalPowerSeriesFunctions examples
====================================================================

This package has no description

See Also:
   o )show UnivariateFormalPowerSeriesFunctions

---
UnivariateFormalPowerSeriesFunctions (UFPS1)

Exports:
hadamard

— package UFPS1 UnivariateFormalPowerSeriesFunctions —

)abbrev package UFPS1 UnivariateFormalPowerSeriesFunctions
++ Description:
++ This package has no description

UnivariateFormalPowerSeriesFunctions(Coef: Ring): Exports == Implementation
where

UFPS ==> UnivariateFormalPowerSeries Coef

Exports == with

  hadamard: (UFPS, UFPS) -> UFPS

Implementation == add

  hadamard(f, g) ==
  series map((z1:Coef,z2:Coef):Coef +-> z1*z2,
              coefficients f, coefficients g)
  $StreamFunctions3(Coef, Coef, Coef)

———

— UFPS1.dotabb ——

"UFPS1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UFPS1"]
"UFPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UFPS"]
"UFPS1" -> "UFPS"
package ULS2 UnivariateLaurentSeriesFunctions2

---

---

== UnivariateLaurentSeriesFunctions2.help ==

UnivariateLaurentSeriesFunctions2 examples

Mapping package for univariate Laurent series. This package allows one to apply a function to the coefficients of a univariate Laurent series.

See Also:
- )show UnivariateLaurentSeriesFunctions2

---
UnivariateLaurentSeriesFunctions2 (ULS2)

Exports:
map

— package ULS2 UnivariateLaurentSeriesFunctions2 —

)abbrev package ULS2 UnivariateLaurentSeriesFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 5 March 1990
++ Date Last Updated: 5 March 1990
++ Description:
++ Mapping package for univariate Laurent series
++ This package allows one to apply a function to the coefficients of
++ a univariate Laurent series.

UnivariateLaurentSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):_
Exports == Implementation where
Coef1 : Ring
Coef2 : Ring
var1: Symbol
var2: Symbol
cen1: Coef1
cen2: Coef2
ULS1 ==> UnivariateLaurentSeries(Coef1, var1, cen1)
ULS2 ==> UnivariateLaurentSeries(Coef2, var2, cen2)
UTS1 ==> UnivariateTaylorSeries(Coef1, var1, cen1)
UTS2 ==> UnivariateTaylorSeries(Coef2, var2, cen2)
UTSF2 ==> UnivariateTaylorSeriesFunctions2(Coef1, Coef2, UTS1, UTS2)

Exports ==> with
map: (Coef1 -> Coef2,ULS1) -> ULS2
++ \spad{map(f,g(x))} applies the map f to the coefficients of the Laurent
++ series \spad{g(x)}.

Implementation ==> add
map(f, ups) == laurent(degree ups, map(f, taylorRep ups)$UTSF2)

package UPOLYC2 UnivariatePolynomialCategoryFunctions2

--- UnivariatePolynomialCategoryFunctions2.input ---

)set break resume
)sys rm -f UnivariatePolynomialCategoryFunctions2.output
)spool UnivariatePolynomialCategoryFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show UnivariatePolynomialCategoryFunctions2
--E 1

)spool
)lisp (bye)

---

package UPOLYC2 UnivariatePolynomialCategoryFunctions2.help ---

====================================================================
UnivariatePolynomialCategoryFunctions2 examples
====================================================================

Mapping from polynomials over R to polynomials over S given a map from R to S assumed to send zero to zero.
CHAPTER 22. CHAPTER U

See Also:
 o )show UnivariatePolynomialCategoryFunctions2

UnivariatePolynomialCategoryFunctions2 (UPOLYC2)

Exports:
map

--- package UPOLYC2 UnivariatePolynomialCategoryFunctions2 ---

)abbrev package UPOLYC2 UnivariatePolynomialCategoryFunctions2
++ Description:
++ Mapping from polynomials over R to polynomials over S
++ given a map from R to S assumed to send zero to zero.

UnivariatePolynomialCategoryFunctions2(R,PR,S,PS): Exports == Impl where
    R, S: Ring
    PR : UnivariatePolynomialCategory R
    PS : UnivariatePolynomialCategory S

Exports ==
    with
    map: (R -> S, PR) -> PS
    ++ map(f, p) takes a function f from R to S,
    ++ and applies it to each (non-zero) coefficient of a polynomial p
    ++ over R, getting a new polynomial over S.
    ++ Note that since the map is not applied to zero elements, it may map
    ++ zero to zero.

 Impl ==
    add
    map(f, p) ==
ans : PS := 0
while p ^= 0 repeat
   ans := ans + monomial(f leadingCoefficient p, degree p)
   p := reductum p
ans

---

— UPOLYC2.dotabb —

"UPOLYC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPOLYC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPOLYC2" -> "PFECAT"

---

package UPCDEN UnivariatePolynomialCommonDenominator

— UnivariatePolynomialCommonDenominator.input —

)set break resume
)sys rm -f UnivariatePolynomialCommonDenominator.output
)spool UnivariatePolynomialCommonDenominator.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show UnivariatePolynomialCommonDenominator
--E 1

)spool
)lisp (bye)

---

— UnivariatePolynomialCommonDenominator.help —

====================================================================
UnivariatePolynomialCommonDenominator examples
====================================================================


UnivariatePolynomialCommonDenominator provides functions to compute the common denominator of the coefficients of univariate polynomials over the quotient field of a gcd domain.

See Also:
- )show UnivariatePolynomialCommonDenominator

---

UnivariatePolynomialCommonDenominator (UPCDEN)

Exports:
- clearDenominator
- commonDenominator
- splitDenominator

---

)abbrev package UPCDEN UnivariatePolynomialCommonDenominator
+- Author: Manuel Bronstein
+- Date Created: 2 May 1988
+- Date Last Updated: 22 Feb 1990
+- Description:
  + UnivariatePolynomialCommonDenominator provides functions to compute the common denominator of the coefficients of univariate polynomials over the quotient field of a gcd domain.

UnivariatePolynomialCommonDenominator(R, Q, UP):Exports == Impl where
- R : IntegralDomain
- Q : QuotientFieldCategory R
- UP: UnivariatePolynomialCategory Q

Exports == with
- commonDenominator: UP -> R
  + commonDenominator(q) returns a common denominator d for
++ the coefficients of q.

clearDenominator : UP -> UP
++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
++ a common denominator for the coefficients of q.

splitDenominator : UP -> Record(num: UP, den: R)
++ splitDenominator(q) returns \spad{[p, d]} such that \spad{q = p/d} and d
++ is a common denominator for the coefficients of q.

Impl ==> add
import CommonDenominator(R, Q, List Q)

commonDenominator p == commonDenominator coefficients p

clearDenominator p ==
d := commonDenominator p
map(x +-> numer(d*x)::Q, p)

splitDenominator p ==
d := commonDenominator p
[map(x +-> numer(d*x)::Q, p), d]

— UPCDEN.dotabb —

"UPCDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPCDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPCDEN" -> "PFECAT"

— package UPDECOMP UnivariatePolynomialDecompositionPackage —

— UnivariatePolynomialDecompositionPackage.input —

)set break resume
)sys rm -f UnivariatePolynomialDecompositionPackage.output
)spool UnivariatePolynomialDecompositionPackage.output
)set message test on
)set message auto off
)clear all
UnivariatePolynomialDecompositionPackage (UPDECOMP)

Exports:
- leftFactorIfCan
- monicCompleteDecompose
- monicDecomposeIfCan
- monicRightFactorIfCan
- rightFactorIfCan

--- package UPDECOMP UnivariatePolynomialDecompositionPackage ---
)abbrev package UPDECOMP UnivariatePolynomialDecompositionPackage
++ Author: Frederic Lehobey
++ Date Created: 17 June 1996
++ Date Last Updated: 4 June 1997
++ References:
++ [1] Peter Henrici, Automatic Computations with Power Series,
++ Journal of the Association for Computing Machinery, Volume 3, No. 1,
++ January 1956, 10-15
-- Decomposition would be speeded up (O(n log n) instead of O(n^2)) by
++ the Tame Case, Journal of Symbolic Computation (1990) 9, 281-299
++ Formal Power Series, Journal of the Association for Computing
++ Machinery, Vol. 25, No. 4, October 1978, 581-595
++ [5] R. P. Brent, Multiple-Precision Zero-Finding Methods and the
++ Complexity of Elementary Function Evaluation, Analytic
++ Computational Complexity, J. F. Traub, Ed., Academic Press,
++ New York 1975, 151-176
++ Description:
++ UnivariatePolynomialDecompositionPackage implements
++ functional decomposition of univariate polynomial with coefficients
++ in an \spad{IntegralDomain} of \spad{CharacteristicZero}.

UnivariatePolynomialDecompositionPackage(R,UP): Exports == Implementation where
R : Join(IntegralDomain,CharacteristicZero)
UP : UnivariatePolynomialCategory(R)
N ==> NonNegativeInteger
LR ==> Record(left: UP, right: UP)
QR ==> Record(quotient: UP, remainder: UP)

Exports == with

monicRightFactorIfCan: (UP,N) -> Union(UP,"failed")
  ++ monicRightFactorIfCan(f,d) returns a candidate to be the
  ++ monic right factor \( h \) in \( f = g \circ h \) of degree \( d \) of a
  ++ functional decomposition of the polynomial \( f \) or
  ++ \spad{"failed"} if no such candidate.
rightFactorIfCan: (UP,N,R) -> Union(UP,"failed")
  ++ rightFactorIfCan(f,d,c) returns a candidate to be the
  ++ right factor \( h \) in \( f = g \circ h \) of degree \( d \) with leading
  ++ coefficient \( c \) of a functional decomposition of the
  ++ polynomial \( f \) or \spad{"failed"} if no such candidate.
leftFactorIfCan: (UP,UP) -> Union(UP,"failed")
  ++ leftFactorIfCan(f,h) returns the left factor \( g \) in \( f = g \circ h \)
  ++ of the functional decomposition of the polynomial \( f \) with
  ++ given \( h \) or \spad{"failed"} if \( g \) does not exist.
monicDecomposeIfCan: UP -> Union(LR, "failed")
++ monicDecomposeIfCan(f) returns a functional decomposition
++ of the monic polynomial f of "failed" if it has not found any.
monicCompleteDecompose: UP -> List UP
++ monicCompleteDecompose(f) returns a list of factors of f for
++ the functional decomposition ([ f1, ..., fn ] means
++ f = f1 o ... o fn).

Implementation ==> add

rightFactorIfCan(p, dq, lcq) ==
dp := degree p
zero? lcq =>
  error "rightFactorIfCan: leading coefficient may not be zero"
(zero? dp) or (zero? dq) => "failed"
nc := dp exquo dq
nc case "failed" => "failed"
n := nc::N
s := subtractIfCan(dq, 1)::N
lcp := leadingCoefficient p
q: UP := monomial(lcq, dq)
k: N
for k in 1..s repeat
  c: R := 0
  i: N
  for i in 0..subtractIfCan(k, 1)::N repeat
    c := c+(k::R-(n::R+1)*(i::R))*
    coefficient(q, subtractIfCan(dq, i)::N)*
    coefficient(p, subtractIfCan(dp+i, k)::N)
  cquo := c exquo ((k*n)::R*lcp)
  cquo case "failed" => return "failed"
  q := q+monomial(cquo::R, subtractIfCan(dq, k)::N)
q

monicRightFactorIfCan(p, dq) == rightFactorIfCan(p, dq, 1$R)

import UnivariatePolynomialDivisionPackage(R, UP)

leftFactorIfCan(f, h) ==
g: UP := 0
zero? degree h => "failed"
for i in 0.. while not zero? f repeat
  qrf := divideIfCan(f, h)
  qrf case "failed" => return "failed"
  qr := qrf : QR
  r := qr.remainder
  not ground? r => return "failed"
  g := g+monomial(ground(r), i)
  f := qr.quotient
  g
monicDecomposeIfCan f ==
   df := degree f
   zero? df => "failed"
   for dh in 2..subtractIfCan(df,1)::N | zero?(df rem dh) repeat
     h := monicRightFactorIfCan(f,dh)
     h case UP =>
       g := leftFactorIfCan(f,h::UP)
       g case UP => return [g::UP,h::UP]
   "failed"

monicCompleteDecompose f ==
   cf := monicDecomposeIfCan f
   cf case "failed" => [ f ]
   lr := cf :: LR
   append(monicCompleteDecompose lr.left,[lr.right])
UnivariatePolynomialDivisionPackage provides a division for non monic univariate polynomials with coefficients in an IntegralDomain.

See Also:
- )show UnivariatePolynomialDivisionPackage

Exports:
- divideIfCan

---

UnivariatePolynomialDivisionPackage (UPDIVP)

---

)abbrev package UPDIVP UnivariatePolynomialDivisionPackage
++ Author: Frederic Lehobey
++ Date Created: 3 June 1997
++ Date Last Updated: 3 June 1997
++ Description:
++ UnivariatePolynomialDivisionPackage provides a
++ division for non monic univariate polynomials with coefficients in
++ an \spad{IntegralDomain}.

UnivariatePolynomialDivisionPackage(R,UP): Exports == Implementation where
R : IntegralDomain
UP : UnivariatePolynomialCategory(R)
N ==> NonNegativeInteger
QR ==> Record(quotient: UP, remainder: UP)

Exports ==> with

divideIfCan: (UP,UP) -> Union(QR,"failed")
++ divideIfCan(f,g) returns quotient and remainder of the
++ division of f by g or "failed" if it has not succeeded.

Implementation ==> add

divideIfCan(p1:UP,p2:UP):Union(QR,"failed") ==
zero? p2 => error "divideIfCan: division by zero"
-- one? (lc := leadingCoefficient p2) => monicDivide(p1,p2)
((lc := leadingCoefficient p2) = 1) => monicDivide(p1,p2)
q: UP := 0
while not ((e := subtractIfCan(degree(p1),degree(p2))) case "failed") repeat
  c := leadingCoefficient(p1) exquo lc
  c case "failed" => return "failed"
  ee := e::N
  q := q+monomial(c::R,ee)
  p1 := p1-c*mapExponents(x +-> x+ee, p2)
[q,p1]
CHAPTER 22. CHAPTER U

)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialFunctions2
--E 1

)spool
)lisp (bye)

== UnivariatePolynomialFunctions2.help ==

UnivariatePolynomialFunctions2 examples
====================================================================
This package lifts a mapping from coefficient rings R to S to a
mapping from UnivariatePolynomial(x,R) to UnivariatePolynomial(y,S).

Note that the mapping is assumed to send zero to zero, since it will
only be applied to the non-zero coefficients of the polynomial.

See Also:
o )show UnivariatePolynomialFunctions2

UnivariatePolynomialFunctions2 (UP2)

Exports:
map
package UP2 UnivariatePolynomialFunctions2

)abbrev package UP2 UnivariatePolynomialFunctions2
+ Description:
+ This package lifts a mapping from coefficient rings R to S to
+ a mapping from \spadtype{UnivariatePolynomial}(x,R) to
+ \spadtype{UnivariatePolynomial}(y,S). Note that the mapping is assumed
+ to send zero to zero, since it will only be applied to the non-zero
+ coefficients of the polynomial.

UnivariatePolynomialFunctions2(x:Symbol, R:Ring, y:Symbol, S:Ring): with
  map: (R -> S, UnivariatePolynomial(x,R)) -> UnivariatePolynomial(y,S)
  ++ map(func, poly) creates a new polynomial by applying func to
  ++ every non-zero coefficient of the polynomial poly.
  == add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
    UnivariatePolynomial(x, R), S, UnivariatePolynomial(y, S))

package UPMP UnivariatePolynomialMultiplicationPackage

— UnivariatePolynomialMultiplicationPackage.input —

)set break resume
)sys rm -f UnivariatePolynomialMultiplicationPackage.output
)spool UnivariatePolynomialMultiplicationPackage.output
)set message test on
)set message auto off
)clear all
This package implements Karatsuba’s trick for multiplying (large) univariate polynomials. It could be improved with a version doing the work on place and also with a special case for squares. We’ve done this in Basicmath, but we believe that this out of the scope of Axiom.

See Also:
  o )show UnivariatePolynomialMultiplicationPackage

---

UnivariatePolynomialMultiplicationPackage (UPMP)

Exports:
  karatsuba  karatsubaOnce  noKaratsuba

---

package UPMP UnivariatePolynomialMultiplicationPackage ---

)abbrev package UPMP UnivariatePolynomialMultiplicationPackage
++ Author: Marc Moreno Maza
++ Date Created: 14.08.2000
++ Description:
++ This package implements Karatsuba’s trick for multiplying
++ (large) univariate polynomials. It could be improved with
++ a version doing the work on place and also with a special
++ case for squares. We’ve done this in Basicmath, but we
++ believe that this out of the scope of AXIOM.

UnivariatePolynomialMultiplicationPackage(R: Ring, U: UnivariatePolynomialCategory(R)): C == T

where
HL => Record(quotient:U, remainder:U)
C == with
noKaratsuba: (U, U) -> U
++ \spad{noKaratsuba(a,b)} returns \spad{a*b} without
++ using Karatsuba’s trick at all.
karatsubaOnce: (U, U) -> U
++ \spad{karatsubaOnce(a,b)} returns \spad{a*b} by applying
++ Karatsuba’s trick once. The other multiplications
++ are performed by calling \spad{\ast} from \spad{U}.
karatsuba: (U, U, NonNegativeInteger, NonNegativeInteger) -> U;
++ \spad{karatsuba(a,b,l,k)} returns \spad{a*b} by applying
++ Karatsuba’s trick provided that both \spad{a} and \spad{b}
++ have at least \spad{l} terms and \spad{k > 0} holds
++ and by calling \spad{noKaratsuba} otherwise. The other
++ multiplications are performed by recursive calls with
++ the same third argument and \spad{k-1} as fourth argument.

T == add
noKaratsuba(a,b) ==
zero? a => a
zero? b => b
zero?(deg(a)) => leadingCoefficient(a) * b
zero?(deg(b)) => a * leadingCoefficient(b)
lu: List(U) := reverse monomials(a)
res: U := 0;
for u in lu repeat
  res := pomopo!(res, leadingCoefficient(u), deg(u), b)
res
karatsubaOnce(a:U,b:U): U ==
da := minimumDegree(a)
db := minimumDegree(b)
if not zero? da then a := shiftRight(a,da)
if not zero? db then b := shiftRight(b,db)
d := da + db
n: NonNegativeInteger := min(deg(a),deg(b)) quo 2
rec: HL := karatsubaDivide(a, n)
ha := rec.quotient
la := rec.remainder
rec := karatsubaDivide(b, n)
hb := rec.quotient
lb := rec.remainder
w: U := (ha - la) * (lb - hb)
u: U := la * lb
v: U := ha * hb
w := w + (u + v)
w := shiftLeft(w,n) + u
zero? d => shiftLeft(v,2*n) + w
shiftLeft(v,2*n + d) + shiftLeft(w,d)
karatsuba(a:U,b:U,l:NonNegativeInteger,k:NonNegativeInteger): U ==
zero? k => noKaratsuba(a,b)
degree(a) < l => noKaratsuba(a,b)
degree(b) < l => noKaratsuba(a,b)
numberOfMonomials(a) < l => noKaratsuba(a,b)
numberOfMonomials(b) < l => noKaratsuba(a,b)
da := minimumDegree(a)
db := minimumDegree(b)
if not zero? da then a := shiftRight(a,da)
if not zero? db then b := shiftRight(b,db)
d := da + db
n: NonNegativeInteger := min(degree(a),degree(b)) quo 2
k := subtractIfCan(k,1)::NonNegativeInteger
rec: HL := karatsubaDivide(a, n)
ha := rec.quotient
la := rec.remainder
rec := karatsubaDivide(b, n)
hb := rec.quotient
lb := rec.remainder
w: U := karatsuba(ha - la, lb - hb, l, k)
u: U := karatsuba(la, lb, l, k)
v: U := karatsuba(ha, hb, l, k)
w := w + (u + v)
w := shiftLeft(w,n) + u
zero? d => shiftLeft(v,2*n) + w
shiftLeft(v,2*n + d) + shiftLeft(w,d)
package UPSQFREE UnivariatePolynomialSquareFree

---

UnivariatePolynomialSquareFree examples

This package provides for square-free decomposition of univariate polynomials over arbitrary rings, i.e. a partial factorization such that each factor is a product of irreducibles with multiplicity one and the factors are pairwise relatively prime. If the ring has characteristic zero, the result is guaranteed to satisfy this condition. If the ring is an infinite ring of finite characteristic, then it may not be possible to decide when polynomials contain factors which are $p$th powers. In this case, the flag associated with that polynomial is set to "nil" (meaning that that polynomials are not guaranteed to be square-free).

See Also:
- )show UnivariatePolynomialSquareFree
UnivariatePolynomialSquareFree (UPSQFREE)

Exports:
  squareFree   squareFreePart   BumInSepFFE

— package UPSQFREE UnivariatePolynomialSquareFree —

)abbrev package UPSQFREE UnivariatePolynomialSquareFree
++ Author: Dave Barton, Barry Trager
++ Description:
++ This package provides for square-free decomposition of
++ univariate polynomials over arbitrary rings, i.e.
++ a partial factorization such that each factor is a product
++ of irreducibles with multiplicity one and the factors are
++ pairwise relatively prime. If the ring
++ has characteristic zero, the result is guaranteed to satisfy
++ this condition. If the ring is an infinite ring of
++ finite characteristic, then it may not be possible to decide when
++ polynomials contain factors which are pth powers. In this
++ case, the flag associated with that polynomial is set to "nil"
++ (meaning that that polynomials are not guaranteed to be square-free).

UnivariatePolynomialSquareFree(RC: IntegralDomain, P): C == T

where
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FF ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

P:Join(UnivariatePolynomialCategory(RC), IntegralDomain) with
  gcd: (%, %) -> %
  ++ gcd(p, q) computes the greatest-common-divisor of p and q.

C == with
  squareFree: P -> Factored(P)
  ++ squareFree(p) computes the square-free factorization of the
  ++ univariate polynomial p. Each factor has no repeated roots, and the
  ++ factors are pairwise relatively prime.
  squareFreePart: P -> P
  ++ squareFreePart(p) returns a polynomial which has the same
++ irreducible factors as the univariate polynomial p, but each
++ factor has multiplicity one.

BumInSepFFE: FF -> FF
++ BumInSepFFE(f) is a local function, exported only because
++ it has multiple conditional definitions.

T == add

if RC has CharacteristicZero then
  squareFreePart(p:P) == (p exquo gcd(p, differentiate p))::P
else
  squareFreePart(p:P) ==
    unit(s := squareFree(p)$%) * */[f.factor for f in factors s]

if RC has FiniteFieldCategory then
  BumInSepFFE(ffe:FF) ==
    ["sqfr", map(charthRoot, ffe.fctr), characteristic$P*ffe.xpnt]
else if RC has CharacteristicNonZero then
  BumInSepFFE(ffe:FF) ==
    np := multiplyExponents(ffe.fctr, characteristic$P:NonNegativeInteger)
    (nthrp := charthRoot(np)) case "failed" =>
      ["nil", np, ffe.xpnt]
    ["sqfr", nthrp, characteristic$P*ffe.xpnt]
else
  BumInSepFFE(ffe:FF) ==
    ["nil",
      multiplyExponents(ffe.fctr, characteristic$P:NonNegativeInteger),
      ffe.xpnt]

if RC has CharacteristicZero then
  squareFree(p:P) ==
    --Yun's algorithm - see SYMSAC '76, p.27
    --Note ci primitive is, so GCD's don't need to %do contents.
    --Change gcd to return cofctrs also?
    ci:=p; di:=differentiate(p); pi:=gcd(ci,di)
    degree(pi)=0 =>
      (u,c,a):=unitNormal(p)
      makeFR(u,[["sqfr",c,a]])
    i:NonNegativeInteger:=0; lffe:List FF:=[]
    lcp := leadingCoefficient p
    while degree(ci)^=0 repeat
      ci:=(ci exquo pi)::P
      di:=(di exquo pi)::P - differentiate(ci)
      pi:=gcd(ci,di)
      i:=i+1
      degree(pi) > 0 =>
        lcp:=(lcp exquo (leadingCoefficient(pi)**i))::RC
        lffe:=[[["sqfr",pi,i],lffe]
      makeFR(lcp::P,lffe)
else
squareFree(p:P) == --Musser's algorithm - see SYMSAC '76, p.27
--p MUST BE PRIMITIVE, Any characteristic.
--Note ci primitive, so GCD's don't need to %do contents.
--Change gcd to return cofctrs also?
  ci := gcd(p,differentiate(p))
degree(ci)=0 =>
    (u,c,a):=unitNormal(p)
    makeFR(u,[["sqfr",c,1]])
  di := (p exquo ci)::P
  i:NonNegativeInteger:=0; lffe:List FF:=[]
dunit : P := 1
while degree(di)^=0 repeat
  diprev := di
di := gcd(ci,di)
ci:=(ci exquo di)::P
  i:=i+1
degree(diprev) = degree(di) =>
    lc := (leadingCoefficient(diprev) exquo leadingCoefficient(di))::RC
dunit := lc**i * dunit
  pi:=(diprev exquo di)::P
  lffe:=["sqfr",pi,i],lffe]
dunit := dunit * di ** (i+1)
degree(ci)=0 => makeFR(dunit*ci,lffe)
redSqfr:=squareFree(divideExponents(ci,characteristic$P)::P)
lsnil:= [BumInSepFFE(ffe) for ffe in factorList redSqfr]
lffe:=append(lsnil,lffe)
makeFR(dunit*(unit redSqfr),lffe)

---

[U H SQFREE dotabb]
"UPSQFREE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPSQFREE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPSQFREE" -> "PFECAT"

---

package UPXS2 UnivariatePuiseuxSeriesFunctions2

--- UnivariatePuiseuxSeriesFunctions2.input ---

)set break resume
This package allows one to apply a function to the coefficients of a univariate Puiseux series.

See Also:
- )show UnivariatePuiseuxSeriesFunctions2

UnivariatePuiseuxSeriesFunctions2 (UPXS2)

Exports:
- map
package UPXS2 UnivariatePuiseuxSeriesFunctions2

)abbrev package UPXS2 UnivariatePuiseuxSeriesFunctions2
++ Author: Scott C. Morrison
++ Date Created: 5 April 1991
++ Date Last Updated: 5 April 1991
++ Description:
++ Mapping package for univariate Puiseux series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Puiseux series.

UnivariatePuiseuxSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):_
Exports == Implementation where
  Coef1 : Ring
  Coef2 : Ring
  var1: Symbol
  var2: Symbol
  cen1: Coef1
  cen2: Coef2
  UPS1 ==> UnivariatePuiseuxSeries(Coef1, var1, cen1)
  UPS2 ==> UnivariatePuiseuxSeries(Coef2, var2, cen2)
  ULSP2 ==> UnivariateLaurentSeriesFunctions2(Coef1, Coef2, var1, var2, cen1, cen2)

Exports ==> with
  map: (Coef1 -> Coef2,UPS1) -> UPS2
  ++ \spad{map(f,g(x))} applies the map f to the coefficients of the
  ++ Puiseux series \spad{g(x)}.

Implementation ==> add
  map(f,ups) == puiseux(rationalPower ups, map(f, laurentRep ups)$ULSP2)

———

UPXS2.dotabbn
package OREPCTO UnivariateSkewPolynomialCategoryOps

--- UnivariateSkewPolynomialCategoryOps.input ---

)set break resume
)sys rm -f UnivariateSkewPolynomialCategoryOps.output
)spool UnivariateSkewPolynomialCategoryOps.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateSkewPolynomialCategoryOps
--E 1

)spool
)lisp (bye)

---

--- UnivariateSkewPolynomialCategoryOps.help ---

================================================================================
UnivariateSkewPolynomialCategoryOps examples
================================================================================

UnivariateSkewPolynomialCategoryOps provides products and divisions of univariate skew polynomials.

See Also:
  o )show UnivariateSkewPolynomialCategoryOps
UnivariateSkewPolynomialCategoryOps (OREPCTO)

Exports:
apply   leftDivide   monicLeftDivide
monicRightDivide rightDivide times

— package OREPCTO UnivariateSkewPolynomialCategoryOps —

)abbrev package OREPCTO UnivariateSkewPolynomialCategoryOps
++ Author: Manuel Bronstein
++ Date Created: 1 February 1994
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{UnivariateSkewPolynomialCategoryOps} provides products and
++ divisions of univariate skew polynomials.
++ -- Putting those operations here rather than defaults in OREPCAT allows
++ -- OREPCAT to be defined independently of sigma and delta.
++ -- MB 2/94

UnivariateSkewPolynomialCategoryOps(R, C):Exports == Implementation where
R: Ring
C: UnivariateSkewPolynomialCategory R

N ==> NonNegativeInteger
MOR ==> Automorphism R
QUOREM ==> Record(quotient: C, remainder: C)

Exports == with
times: (C, C, MOR, R -> R) -> C
++ times(p, q, sigma, delta) returns \spad{p * q}.
++ \spad{\sigma} and \spad{\delta} are the maps to use.
apply: (C, R, R, MOR, R -> R) -> R
++ apply(p, c, m, sigma, delta) returns \spad{p(m)} where the action
++ is given by \spad{x m = c sigma(m) + delta(m)}.
if R has IntegralDomain then
monicLeftDivide: (C, C, MOR) -> QUOREM
++ monicLeftDivide(a, b, sigma) returns the pair \spad{[q,r]}
such that \( a = b \cdot q + r \) and the degree of \( r \) is
less than the degree of \( b \).
\( b \) must be monic.
This process is called 'left division'.
\( \sigma \) is the morphism to use.

\[
\text{monicRightDivide}(a, b, \sigma) \quad \text{returns the pair } [q, r]\text{ such that } a = q \cdot b + r \text{ and the degree of } r \text{ is less than the degree of } b.
\( b \) must be monic.
This process is called 'right division'.
\( \sigma \) is the morphism to use.

\[
\text{leftDivide}(a, b, \sigma) \quad \text{returns the pair } [q, r]\text{ such that } a = b \cdot q + r \text{ and the degree of } r \text{ is less than the degree of } b.
\( b \) must be monic.
This process is called 'left division'.
\( \sigma \) is the morphism to use.

\[
\text{rightDivide}(a, b, \sigma) \quad \text{returns the pair } [q, r]\text{ such that } a = q \cdot b + r \text{ and the degree of } r \text{ is less than the degree of } b.
\( b \) must be monic.
This process is called 'right division'.
\( \sigma \) is the morphism to use.

Implementation == add

\[
\text{termPoly}(a, n, y, \sigma, \delta) ==
\begin{align*}
\text{zero? } y & \Rightarrow 0 \\
z &: = 0 \\
\text{while } y \neq 0 \text{ repeat} \\
\quad z &: = z + \text{termPoly}(\text{leadingCoefficient } x, \text{degree } x, y, \sigma, \delta) \\
\quad x &: = \text{reductum } x \\
z
\end{align*}
\]

\[
\text{termPoly}(a, n, y, \sigma, \delta) ==
\begin{align*}
\text{zero? } y & \Rightarrow 0 \\
(u &: = \text{subtractIfCan}(n, 1)) \text{ case "failed" } \Rightarrow a \cdot y \\
n1 &: = u::N \\
z &: = 0 \\
\text{while } y \neq 0 \text{ repeat} \\
\quad m &: = \text{degree } y \\
b &: = \text{leadingCoefficient } y \\
z &: = z + \text{termPoly}(a, n1, \text{monomial } (\sigma b, m + 1), \sigma, \delta) \\
\quad + \text{termPoly}(a, n1, \text{monomial } (\delta b, m), \sigma, \delta) \\
y &: = \text{reductum } y
\end{align*}
\]
apply(p, c, x, sigma, delta) ==
  w:R := 0
  xn:R := x
  for i in 0..degree p repeat
    w := w + coefficient(p, i) * xn
    xn := c * sigma xn + delta xn
  w

-- localLeftDivide(a, b) returns [q, r] such that a = q b + r
-- b1 is the inverse of the leadingCoefficient of b
localLeftDivide(a, b, sigma, b1) ==
  zero? b => error "leftDivide: division by 0"
  zero? a or
  (n := subtractIfCan(degree(a),(m := degree b))) case "failed" =>
    [0,a]
  q := monomial((sigma**(-m))(b1 * leadingCoefficient a), n::N)
  qr := localLeftDivide(a - b * q, b, sigma, b1)
  [q + qr.quotient, qr.remainder]

-- localRightDivide(a, b) returns [q, r] such that a = q b + r
-- b1 is the inverse of the leadingCoefficient of b
localRightDivide(a, b, sigma, b1) ==
  zero? b => error "rightDivide: division by 0"
  zero? a or
  (n := subtractIfCan(degree(a),(m := degree b))) case "failed" =>
    [0,a]
  q := monomial(leadingCoefficient(a) * (sigma**n) b1, n::N)
  qr := localRightDivide(a - q * b, b, sigma, b1)
  [q + qr.quotient, qr.remainder]

if R has IntegralDomain then
  monicLeftDivide(a, b, sigma) ==
    unit?(u := leadingCoefficient b) =>
      localLeftDivide(a, b, sigma, recip(u)::R)
    error "monicLeftDivide: divisor is not monic"

monicRightDivide(a, b, sigma) ==
  unit?(u := leadingCoefficient b) =>
    localRightDivide(a, b, sigma, recip(u)::R)
  error "monicRightDivide: divisor is not monic"

if R has Field then
  leftDivide(a, b, sigma) ==
    localLeftDivide(a, b, sigma, inv leadingCoefficient b)

rightDivide(a, b, sigma) ==
  localRightDivide(a, b, sigma, inv leadingCoefficient b)
package UTS2 UnivariateTaylorSeriesFunctions2

UnivariateTaylorSeriesFunctions2.examples

Mapping package for univariate Taylor series. This package allows one to apply a function to the coefficients of a univariate Taylor series.

See Also:
  o )show UnivariateTaylorSeriesFunctions2
UnivariateTaylorSeriesFunctions2 (UTS2)

Exports:
map

— package UTS2 UnivariateTaylorSeriesFunctions2 —

)abbrev package UTS2 UnivariateTaylorSeriesFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 9 February 1990
++ Date Last Updated: 9 February 1990
++ Description:
++ Mapping package for univariate Taylor series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Taylor series.

UnivariateTaylorSeriesFunctions2(Coef1,Coef2,UTS1,UTS2): _
Exports == Implementation where
Coef1 : Ring
Coef2 : Ring
UTS1 : UnivariateTaylorSeriesCategory Coef1
UTS2 : UnivariateTaylorSeriesCategory Coef2
ST2 ==> StreamFunctions2(Coef1,Coef2)

Exports ==> with
map: (Coef1 -> Coef2,UTS1) -> UTS2
++\spad{map(f,g(x))} applies the map f to the coefficients of
++ the Taylor series \spad{g(x)}.

Implementation ==> add

map(f,uts) == series map(f,coefficients uts)$ST2
package UTSODE UnivariateTaylorSeriesODESolver

---

UnivariateTaylorSeriesODESolver.help

This package provides Taylor series solutions to regular linear or non-linear ordinary differential equations of arbitrary order.

See Also:
- )show UnivariateTaylorSeriesODESolver
UnivariateTaylorSeriesODESolver (UTSODE)

Exports:
fixedPointExquo npsode ode ode1 ode2
stFunc1 stFunc2 stFuncN

— package UTSODE UnivariateTaylorSeriesODESolver —

)abbrev package UTSODE UnivariateTaylorSeriesODESolver
++ Author: Stephen Watt (revised by Clifton J. Williamson)
++ Date Created: February 1988
++ Date Last Updated: 30 September 1993
++ Description:
++ Taylor series solutions of explicit ODE's.
++ This package provides Taylor series solutions to regular
++ linear or non-linear ordinary differential equations of
++ arbitrary order.

UnivariateTaylorSeriesODESolver(Coef,UTS):_
Exports == Implementation where
Coef : Algebra Fraction Integer
UTS : UnivariateTaylorSeriesCategory Coef
L ==> List
L2 ==> ListFunctions2
FN ==> (L UTS) -> UTS
ST ==> Stream Coef
YS ==> Y$ParadoxicalCombinatorsForStreams(Coef)
STT ==> StreamTaylorSeriesOperations(Coef)

Exports => with
stFunc1: (UTS -> UTS) -> (ST -> ST)
++ stFunc1(f) is a local function exported due to compiler problem.
++ This function is of no interest to the top-level user.
stFunc2: ((UTS,UTS) -> UTS) -> ((ST,ST) -> ST)
++ stFunc2(f) is a local function exported due to compiler problem.
++ This function is of no interest to the top-level user.
stFuncN: FN -> ((L ST) -> ST)
++ stFuncN(f) is a local function exported due to compiler problem.
++ This function is of no interest to the top-level user.

fixedPointExquo: (UTS, UTS) -> UTS
++ fixedPointExquo((f,g) computes the exact quotient of \spad{f} and
++ \spad{g} using a fixed point computation.

ode1: ((UTS -> UTS), Coef) -> UTS
++ ode1((f,c) is the solution to \spad{y' = f(y)}
++ such that \spad{y(a) = c}.

ode2: ((UTS, UTS) -> UTS, Coef, Coef) -> UTS
++ ode2((f,c0,c1) is the solution to \spad{y'' = f(y,y')} such that
++ \spad{y(a) = c0} and \spad{y'(a) = c1}.

ode: (FN, List Coef) -> UTS
++ ode((f,cl) is the solution to \spad{y<n>=f(y,y',..,y<n-1>)} such that
++ \spad{y<i>(a) = cl.i} for i in 1..n.

mpsode: (L Coef, L FN) -> L UTS
++ mpsode(r,f) solves the system of differential equations
++ \spad{dy[i]/dx =f[i] [x,y[1],y[2],...,y[n]]},
++ \spad{y[i](a) = r[i]} for i in 1..n.

Implementation ==> add

stFunc1 f == s +-> coefficients f series(s)
stFunc2 f == (s1,s2) +-> coefficients f(series(s1),series(s2))
stFuncN f == ls +-> coefficients f map(series,ls)$ListFunctions2(ST,UTS)

import StreamTaylorSeriesOperations(Coef)
divloopre: (Coef, ST, Coef, ST, ST) -> ST
divloopre(hx, tx, hy, ty, c) == delay(concat(hx*hy, hy*(tx-(ty*c))))
divloop: (Coef, ST, Coef, ST) -> ST
divloop(hx, tx, hy, ty) == YS(s +-> divloopre(hx, tx, hy, ty, s))

sdiv: (ST, ST) -> ST
sdiv(x, y) == delay
  empty? x => empty()
  empty? y => error "stream division by zero"
  hx := frst x; tx := rst x
  hy := frst y; ty := rst y
  zero? hy =>
    error "stream division by zero"
  rhy := recip hy
  rhy case "failed" => error "stream division: no reciprocal"
  divloop(hx, tx, rhy::Coef, ty)

fixedPointExquo(f, g) == series sdiv(coefficients f, coefficients g)

-- first order

ode1re: (ST -> ST, Coef, ST) -> ST
ode1re(f, c, y) == lazyIntegrate(c, f y)$STT
iOde1: ((ST -> ST),Coef) -> ST
iOde1(f,c) == YS(s +-> ode1re(f,c,s))

ode1(f,c) == series iOde1(stFunc1 f,c)

-- second order

ode2re: ((ST,ST)-> ST,Coef,Coef,ST) -> ST
ode2re(f,c0,c1,y)==
yi := lazyIntegrate(c1,f(y,deriv(y)$STT))$STT
lazyIntegrate(c0,yi)$STT

iOde2: ((ST,ST) -> ST,Coef,Coef) -> ST
iOde2(f,c0,c1) == YS(s +-> ode2re(f,c0,c1,s))

ode2(f,c0,c1) == series iOde2(stFunc2 f,c0,c1)

-- nth order

odeNre: (List ST -> ST,List Coef,List ST) -> List ST
odeNre(f,cl,yl) ==
  -- yl is [y, y', ..., y<n>]
  -- integrate [y',...,y<n>] to get [y,...,y<n-1>]
  yil := [lazyIntegrate(c,y)$STT for c in cl for y in rest yl]
  -- use y<n> = f(y,...,y<n-1>)
  concat(yil,[f yil])

iOde: ((L ST) -> ST,List Coef) -> ST
iOde(f,cl) == first YS(ls +-> odeNre(f,cl,ls),#cl + 1)

ode(f,cl) == series iOde(stFuncN f,cl)

simulre:(L Coef,L ((L ST) -> ST),L ST) -> L ST
simulre(cst,lsf,c) ==
  [lazyIntegrate(csti,lsfi concat(monom(1,1)$STT,c))
    for csti in cst for lsfi in lsf]

iMpsode:(L Coef,L ((L ST) -> ST)) -> L ST
iMpsode(cs,lsts) == YS(ls +-> simulre(cs,lsts,ls),# cs)

mpsode(cs,lsts) ==
  -- stSol := iMpsode(cs,map(stFuncN,lsts)$L2(FN,(L ST) -> ST))
  stSol := iMpsode(cs,[stFuncN(lst) for lst in lsts])
  map(series,stSol)$L2(ST,UTS)
package UNISEG2 UniversalSegmentFunctions2

— UniversalSegmentFunctions2.input —

)set break resume
)sys rm -f UniversalSegmentFunctions2.output
)spool UniversalSegmentFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UniversalSegmentFunctions2
--E 1

)spool
)lisp (bye)

— UniversalSegmentFunctions2.help —

====================================================================
UniversalSegmentFunctions2 examples
====================================================================

This package provides operations for mapping functions onto segments.

See Also:
  o )show UniversalSegmentFunctions2
UniversalSegmentFunctions2 (UNISEG2)

Exports:
map

— package UNISEG2 UniversalSegmentFunctions2 —

)abbrev package UNISEG2 UniversalSegmentFunctions2
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations for mapping functions onto segments.

UniversalSegmentFunctions2(R:Type, S:Type): with
  map: (R -> S, UniversalSegment R) -> UniversalSegment S
  ++ map(f,seg) returns the new segment obtained by applying
  ++ f to the endpoints of seg.

  if R has OrderedRing then
  map: (R -> S, UniversalSegment R) -> Stream S
  ++ map(f,s) expands the segment s, applying \spad{f} to each value.

== add
map(f:R -> S, u:UniversalSegment R):UniversalSegment S ==
  s := f lo u
  hasHi u => segment(s, f hi u)
  segment s

  if R has OrderedRing then
  map(f:R -> S, u:UniversalSegment R): Stream S ==
  map(f, expand u)$StreamFunctions2(R, S)

— UNISEG2.dotabb —
package UDPO UserDefinedPartialOrdering

--- UserDefinedPartialOrdering.input ---

)set break resume
)sys rm -f UserDefinedPartialOrdering.output
)spool UserDefinedPartialOrdering.output
)set message test on
)set message auto off
)clear all

--$ 1 of 1
)show UserDefinedPartialOrdering
--E 1

)spool
)lisp (bye)

---

--- UserDefinedPartialOrdering.help ---

====================================================================
UserDefinedPartialOrdering examples
====================================================================

Provides functions to force a partial ordering on any set.

See Also:
o )show UserDefinedPartialOrdering

---
CHAPTER 22. CHAPTER U

UserDefinedPartialOrdering (UDPO)

Exports:
  setOrder userOrdered? getOrder largest less? more?

— package UDPO UserDefinedPartialOrdering —

)abbrev package UDPO UserDefinedPartialOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
  ++ Provides functions to force a partial ordering on any set.

UserDefinedPartialOrdering(S:SetCategory): with
  setOrder : List S -> Void
    ++ setOrder([a1,...,an]) defines a partial ordering on S given by:
    ++  (1) \spad{a1 < a2 < ... < an}.
    ++  (2) \spad{b < ai} for \(i = 1..n\) and b not among the ai’s.
    ++  (3) undefined on \spad{(b, c)} if neither is among the ai’s.
  setOrder : (List S, List S) -> Void
    ++ setOrder([b1,...,bm], [a1,...,an]) defines a partial
    ++ ordering on S given by:
    ++  (1) \spad{b1 < b2 < ... < bm < a1 < a2 < ... < an}.
    ++  (2) \spad{bj < c < ai} for c not among the ai’s and bj’s.
    ++  (3) undefined on \spad{(c,d)} if neither is among the ai’s,bj’s.
  getOrder : () -> Record(low: List S, high: List S)
    ++ getOrder() returns \spad{[[b1,...,bm], [a1,...,an]]} such that the
    ++ partial ordering on S was given by
    ++ \spad{setOrder([b1,...,bm], [a1,...,an])}.
  less?: (S, S) -> Union(Boolean, "failed")
    ++ less?(a, b) compares \spad{a} and b in the partial ordering induced by
    ++ setOrder.
  less?: (S, S, (S, S) -> Boolean) -> Boolean
    ++ less?(a, b, fn) compares \spad{a} and b in the partial ordering induced
    ++ by setOrder, and returns \spad{fn(a, b)} if \spad{a}
    ++ and b are not comparable
++ in that ordering.
largest : (List S, (S, S) \rightarrow Boolean) \rightarrow S
++ largest(l, fn) returns the largest element of l where the partial
++ ordering induced by setOrder is completed into a total one by fn.
userOrdered?: () \rightarrow Boolean
++ userOrdered?() tests if the partial ordering induced by
++ setOrder is not empty.
if S has OrderedSet then
  largest: List S \rightarrow S
  ++ largest l returns the largest element of l where the partial
  ++ ordering induced by setOrder is completed into a total one by
  ++ the ordering on S.
more? : (S, S) \rightarrow Boolean
++ more?(a, b) compares a and b in the partial ordering induced
++ by setOrder, and uses the ordering on S if a and b are not
++ comparable in the partial ordering.

== add
llow : Reference List S := ref nil()
lhigh: Reference List S := ref nil()
userOrdered?() == not(empty? deref llow) or not(empty? deref lhigh)
getOrder() == [deref llow, deref lhigh]
setOrder l == setOrder(nil(), l)

setOrder(l, h) ==
  setref(llow, removeDuplicates l)
  setref(lhigh, removeDuplicates h)
void

less?(a, b, f) ==
  (u := less?(a, b)) case "failed" => f(a, b)
u::Boolean

largest(x, f) ==
  empty? x => error "largest: empty list"
  empty? rest x => first x
  a := largest(rest x, f)
  less?(first x, a, f) => a
  first x

less?(a, b) ==
  for x in deref llow repeat
    x = a => return(a ^= b)
    x = b => return false
  aa := bb := false$Boolean
  for x in deref lhigh repeat
    if x = a then
      bb => return false
    aa := true
if x = b then
   aa => return(a ^= b)
   bb := true
aa => false
bb => true
"failed"

if S has OrderedSet then
   more?(a, b) == not less?(a, b, (y,z) +-> y <$S z)
   largest x  == largest(x, (y,z) +-> y <$S z)

package UDVO UserDefinedVariableOrdering

UserDefinedVariableOrdering.input

UserDefinedVariableOrdering.help
UserDefinedVariableOrdering examples
====================================================================

This package provides functions to allow the user to select the ordering on the variables and operators for displaying polynomials, fractions and expressions. The ordering affects the display only and not the computations.

See Also:
o )show UserDefinedVariableOrdering

UserDefinedVariableOrdering (UDVO)

Exports:
resetVariableOrder getVariableOrder setVariableOrder

--- package UDVO UserDefinedVariableOrdering ---

)abbrev package UDVO UserDefinedVariableOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ This package provides functions to allow the user to select the ordering on the variables and operators for displaying polynomials, fractions and expressions. The ordering affects the display only and not the computations.

UserDefinedVariableOrdering(): with
  setVariableOrder : List Symbol -> Void
    ++ setVariableOrder([a1,...,an]) defines an ordering on the
++ variables given by \spad{a1 > a2 > \ldots > an > other variables}.

setVariableOrder : (List Symbol, List Symbol) -> Void
++ setVariableOrder([b1,\ldots,bm], [a1,\ldots,an]) defines an ordering
++ on the variables given by
++ \spad{b1 > b2 > \ldots > bm > other variables > a1 > a2 > \ldots > an}.

getVariableOrder : () -> Record(high:List Symbol, low:List Symbol)
++ getVariableOrder() returns \spad{[[b1,\ldots,bm], [a1,\ldots,an]]} such that
++ the ordering on the variables was given by
++ \spad{setVariableOrder([b1,\ldots,bm], [a1,\ldots,an])}.

resetVariableOrder: () -> Void
++ resetVariableOrder() cancels any previous use of
++ setVariableOrder and returns to the default system ordering.

== add
import UserDefinedPartialOrdering(Symbol)

setVariableOrder l == setOrder reverse l
setVariableOrder(l1, l2) == setOrder(reverse l2, reverse l1)
resetVariableOrder() == setVariableOrder(nil(), nil())

getVariableOrder() ==
  r := getOrder()
  [reverse(r.high), reverse(r.low)]

— UDVO.dotabb —

"UDVO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UDVO"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"UDVO" -> "BASTYPE"
"UDVO" -> "KOERCE"

package UTSODETL UTSodetools

— UTSodetools.input —

)set break resume
)sys rm -f UTSodetools.output
)spool UTSodetools.output
)set message test on
)set message auto off
 UTsodetools (UTSODETL)

Exports:
LODO2FUN RF2UTS UTS2UP

--- package UTsodetools UTsodetools ---

)abbrev package UTsODETL UTsodetools
++ Author: Manuel Bronstein
++ Date Created: 31 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spad{RUTSodetools} provides tools to interface with the series
++ ODE solver when presented with linear ODEs.

UTSodetools(F, UP, L, UTS): Exports == Implementation where
F : Ring
UP : UnivariatePolynomialCategory F
L : LinearOrdinaryDifferentialOperatorCategory UP
UTS: UnivariateTaylorSeriesCategory F

Exports ==> with
  UP2UTS: UP -> UTS
    ++ UP2UTS(p) converts \spad{p} to a Taylor series.
  UTS2UP: (UTS, NonNegativeInteger) -> UP
    ++ UTS2UP(s, n) converts the first \spad{n} terms of \spad{s}
    ++ to a univariate polynomial.
  LODO2FUN: L -> (List UTS -> UTS)
    ++ LODO2FUN(op) returns the function to pass to the series ODE
    ++ solver in order to solve \spad{op y = 0}.
  if F has IntegralDomain then
    RF2UTS: Fraction UP -> UTS
      ++ RF2UTS(f) converts \spad{f} to a Taylor series.

Implementation ==> add
  fun: (Vector UTS, List UTS) -> UTS

UP2UTS p ==
  q := p(monomial(1, 1) + center(0)::UP)
  +/[monomial(coefficient(q, i), i)$UTS for i in 0..degree q]

UTS2UP(s, n) ==
  xmc ::= monomial(1, 1)$UP - center(0)::UP
  xmcn.UP :: 1
  ans.UP :: 0
  for i in 0..n repeat
    ans ::= ans + coefficient(s, i) * xmcn
    xmcn ::= xmc * xmcn
  ans

LODO2FUN op ==
  a ::= recip(UP2UTS(~ leadingCoefficient op))::UTS
  n ::= (degree(op) - 1)::NonNegativeInteger
  v ::= [a * UP2UTS coefficient(op, i) for i in 0..n]$Vector(UTS)
  r ::= (l1: List UTS): UTS +-> fun(v, l1)
  r

fun(v, 1) ==
ans: UTS := 0
for b in l for i in 1.. repeat ans := ans + v.i * b
ans

if F has IntegralDomain then
RF2UTS f == UP2UTS(numer f) * recip(UP2UTS denom f)::UTS

package POLYVEC U32VectorPolynomialOperations

— U32VectorPolynomialOperations.input —

)set break resume
)sys rm -f U32VectorPolynomialOperations.output
)spool U32VectorPolynomialOperations.output
)set message test on
)set message auto off
)clear all

|--S 1 of 1
)show U32VectorPolynomialOperations
|--E 1

)spool
)lisp (bye)

———

— U32VectorPolynomialOperations.help —

====================================================================
U32VectorPolynomialOperations examples
====================================================================
This is a low-level package which implements operations on vectors treated as univariate modular polynomials. Most operations takes modulus as parameter. Modulus is machine sized prime which should be small enough to avoid overflow in intermediate calculations.

See Also:
- )show U32VectorPolynomialOperations

---

**U32VectorPolynomialOperations (POLYVEC)**

---

)`abbrev package POLYVEC U32VectorPolynomialOperations`  
++ Description:  
++ This is a low-level package which implements operations on vectors treated as univariate modular polynomials. Most operations takes modulus as parameter. Modulus is machine sized prime which should be small enough to avoid overflow in intermediate calculations.  
U32VectorPolynomialOperations() : Export == Implementation where  
   PA ==> U32Vector
PACKAGE POLYVEC U32VECTORPOLYNOMIALOPERATIONS

Export ==> with

copy_first : (PA, PA, Integer) -> Void
++ copy_first(v1, v2, n) copies first n elements
++ of v2 into n first positions in v1.
copy_slice : (PA, PA, Integer, Integer) -> Void
++ copy_first(v1, v2, m, n) copies the slice of v2 starting
++ at m elements and having n elements into corresponding
++ positions in v1.
eval_at : (PA, Integer, Integer, Integer) -> Integer
++ eval_at(v, deg, pt, p) treats v as coefficients of
++ polynomial of degree deg and evaluates the
++ polynomial at point pt modulo p
++
++X a:=new(3,1)$U32VEC
++X a.1:=2
++X eval_at(a,2,3,1024)
++X eval_at(a,2,2,8)
++X eval_at(a,2,3,10)

vector_add_mul : (PA, PA, Integer, Integer, Integer, Integer) -> Void
++ vector_add_mul(v1, v2, m, n, c, p) sets v1(m), ..., v1(n) to corresponding entries in v1 + c*v2
++ modulo p.
mul_by_binomial : (PA, Integer, Integer) -> Void
++ mul_by_binomial(v, pt, p) treats v as a polynomial
++ and multiplies in place this polynomial by binomial (x + pt).
++ Highest coefficient of product is ignored.
mul_by_binomial : (PA, Integer, Integer, Integer) -> Void
++ mul_by_binomial(v, deg, pt, p) treats v as coefficients of polynomial of degree deg and
++ multiplies in place this polynomial by binomial (x + pt).
++ Highest coefficient of product is ignored.
mul_by_scalar : (PA, Integer, Integer, Integer) -> Void
++ mul_by_scalar(v, deg, c, p) treats v as coefficients of polynomial of degree deg and
++ multiplies in place this polynomial by scalar c

mul : (PA, PA, Integer) -> PA
++ Polynomial multiplication.
truncated_multiplication : (PA, PA, Integer, Integer) -> PA
++ truncated_multiplication(x, y, d, p) computes
++ x*y truncated after degree d

truncated_mul_add : (PA, PA, PA, Integer, Integer) -> Void
++ truncated_mul_add(x, y, z, d, p) adds to z
++ the produce x*y truncated after degree d

pow : (PA, PositiveInteger, NonNegativeInteger, Integer) -> PA
++ pow(u, n, d, p) returns u^n truncated after degree d, except if
++ n=1, in which case u itself is returned
differentiate : (PA, Integer) -> PA
++ Polynomial differentiation.
differentiate : (PA, NonNegativeInteger, Integer) -> PA
CHAPTER 22. CHAPTER U

++ Polynomial differentiation.
divide! : (PA, PA, PA, Integer) -> Void
++ Polynomial division.
remainder! : (PA, PA, Integer) -> Void
++ Polynomial remainder
vector_combination : (PA, Integer, PA, Integer, Integer, Integer, Integer, p : Integer) -> Void
++ vector_combination(v1, c1, v2, c2, n, delta, p) replaces
++ first n + 1 entires of v1 by corresponding entries of
++ c1*v1+c2*x^delta*v2 mod p.
to_mod_pa : (SparseUnivariatePolynomial Integer, Integer) -> PA
++ to_mod_pa(s, p) reduces coefficients of polynomial
++ s modulo prime p and converts the result to vector
gcd : (PA, PA, Integer) -> PA
++ gcd(v1, v2, p) computes monic gcd of v1 and v2 modulo p.
gcd : (PrimitiveArray PA, Integer, Integer, Integer) -> PA
++ gcd(a, lo, hi, p) computes gcd of elements
++ a(lo), a(lo+1), ..., a(hi).
lcm : (PrimitiveArray PA, Integer, Integer, Integer) -> PA
++ lcm(a, lo, hi, p) computes lcm of elements
++ a(lo), a(lo+1), ..., a(hi).
degree : PA -> Integer
++ degree(v) is degree of v treated as polynomial
extended_gcd : (PA, PA, Integer) -> List(PA)
++ extended_gcd(v1, v2, p) gives [g, c1, c2] such
++ that g is \spad{gcd(v1, v2, p)}, \spad{g = c1*v1 + c2*v2}
++ and degree(c1) < max(degree(v2) - degree(g), 0) and
++ degree(c2) < max(degree(v1) - degree(g), 1)
resultant : (PA, PA, Integer) -> Integer
++ resultant(v1, v2, p) computes resultant of v1 and v2
++ modulo p.

Implementation ==> add

Qmuladdmod ==> QSMULADDMOD6432$Lisp
Qmuladd ==> QSMULADD6432$Lisp
Qmul ==> QSMULMOD32$Lisp
Qdot2 ==> QSDOT2MOD6432$Lisp
Qrem ==> QSMOD6432$Lisp
modInverse ==> invmod

copy_first(np : PA, op : PA, n : Integer) : Void ==
    ns := n pretend SingleInteger
    for j in 0..(ns - 1) repeat
        np(j) := op(j)

copy_slice(np : PA, op : PA, m : Integer, n : Integer) : Void ==
    ns := m pretend SingleInteger
    ns := n pretend SingleInteger
for j in ms..(ms + ns - 1) repeat
   np(j) := op(j)

eval_at(v : PA, deg : Integer, pt : Integer, 
   p : Integer) : Integer ==
   i : SingleInteger := deg::SingleInteger
   res : Integer := 0
   while not(i < 0) repeat
      res := Qmuladdmod(pt, res, v(i), p)
      i := i - 1
   res

to_mod_pa(s : SparseUnivariatePolynomial Integer, p : Integer) : PA ==
   zero?(s) => new(1, 0)$PA
   n0 := degree(s) pretend SingleInteger
   ncoeffs := new((n0+1) pretend NonNegativeInteger, 0)$PA
   while not(zero?(s)) repeat
      n := degree(s)
      ncoeffs(n) := positiveRemainder(leadingCoefficient(s), p)
      s := reductum(s)
   ncoeffs

vector_add_mul(v1 : PA, v2 : PA, m : Integer, n : Integer, 
   c : Integer, p : Integer) : Void ==
   ms := m pretend SingleInteger
   ns := n pretend SingleInteger
   for i in ms..ns repeat
      v1(i) := Qmuladdmod(c, v2(i), v1(i), p)

mul_by_binomial(v : PA, n : Integer, pt : Integer, 
   p : Integer) : Void ==
   prev_coeff : Integer := 0
   ns := n pretend SingleInteger
   for i in 0..(ns - 1) repeat
      pp := v(i)
      v(i) := Qmuladdmod(pt, pp, prev_coeff, p)
      prev_coeff := pp

mul_by_binomial(v : PA, pt : Integer, 
   p : Integer) : Void ==
   mul_by_binomial(v, #v, pt, p)

mul_by_scalar(v : PA, n : Integer, c : Integer, 
   p : Integer) : Void ==
   ns := n pretend SingleInteger
   for i in 0..ns repeat
      v(i) := Qmul(c, v(i), p)

degree(v : PA) : Integer ==
   n := #v
for i in (n - 1)..0 by -1 repeat
    not(v(i) = 0) => return i

vector_combination(v1 : PA, c1 : Integer, _
    v2 : PA, c2 : Integer, _
    n : Integer, delta : Integer, _
    p : Integer) : Void ==
ns := n pretend SingleInteger
ds := delta pretend SingleInteger
if not(c1 = 1) then
    ns + 1 < ds =>
    for i in 0..ns repeat
        v1(i) := Qmul(v1(i), c1, p)
    for i in 0..(ds - 1) repeat
        v1(i) := Qmul(v1(i), c1, p)
    for i in ds..ns repeat
        v1(i) := Qdot2(v1(i), c1, v2(i - ds), c2, p)
else
    for i in ds..ns repeat
        v1(i) := Qmuladdmod(c2, v2(i - ds), v1(i), p)

divide!(r0 : PA, r1 : PA, res : PA, p: Integer) : Void ==
dr0 := degree(r0) pretend SingleInteger
dr1 := degree(r1) pretend SingleInteger
c0 := r1(dr1)
c0 := modInverse(c0, p)
while not(dr0 < dr1) repeat
    delta := dr0 - dr1
c1 := Qmul(c0, r0(dr0), p)
c1 := p - c1
r0(dr0) := 0
dr0 := dr0 - 1
if dr0 < 0 then break
vector_combination(r0, 1, r1, c1, dr0, delta, p)
while r0(dr0) = 0 repeat
    dr0 := dr0 - 1
if dr0 < 0 then break

remainder!(r0 : PA, r1 : PA, p: Integer) : Void ==
dr0 := degree(r0) pretend SingleInteger
dr1 := degree(r1) pretend SingleInteger
c0 := r1(dr1)
c0 := modInverse(c0, p)
while not(dr0 < dr1) repeat
    delta := dr0 - dr1
c1 := Qmul(c0, r0(dr0), p)
c1 := p - c1
r0(dr0) := 0
dr0 := dr0 - 1
if dr0 < 0 then break
vector_combination(r0, 1, r1, c1, dr0, delta, p)
while r0(dr0) = 0 repeat
  dr0 := dr0 - 1
  if dr0 < 0 then break

gcd(x : PA, y : PA, p : Integer) : PA ==
dr0 := degree(y) pretend SingleInteger
dr1 : SingleInteger
if dr0 < 0 then
  tmpp := x
  x := y
  y := tmpp
dr1 := dr0
dr0 := degree(y) pretend SingleInteger
else
  dr1 := degree(x) pretend SingleInteger
dr0 < 0 => return new(1, 0)$PA
r0 := new((dr0 + 1) pretend NonNegativeInteger, 0)$PA
copy_first(r0, y, dr0 + 1)
dr1 < 0 =>
c := r0(dr0)
c := modInverse(c, p)
mul_by_scalar(r0, dr0, c, p)
return r0
r1 := new((dr1 + 1) pretend NonNegativeInteger, 0)$PA
copy_first(r1, x, dr1 + 1)
while 0 < dr1 repeat
  while not(dr0 < dr1) repeat
    delta := dr0 - dr1
c1 := sub_SI(p, r0(dr0))$Lisp
c0 := r1(dr1)
    if c0 ~= 1 and delta > 30 then
      c0 := modInverse(c0, p)
mul_by_scalar(r1, dr1, c0, p)
c0 := 1
r0(dr0) := 0
dr0 := dr0 - 1
vector_combination(r0, c0, r1, c1, dr0, delta, p)
while r0(dr0) = 0 repeat
  dr0 := dr0 - 1
  if dr0 < 0 then break
tmpp := r0
tmp := dr0
r0 := r1
dr0 := dr1
r1 := tmpp
dr1 := tmp
not(dr1 < 0) =>
r1(0) := 1
return r1
c := r0(dr0)
c := modInverse(c, p)
mul_by_scalar(r0, dr0, c, p)
r0

gcd(a : PrimitiveArray PA, lo : Integer, hi: Integer, p: Integer) ==
res := a(lo)
for i in (lo + 1)..hi repeat
    res := gcd(a(i), res, p)
res

lcm2(v1 : PA, v2 : PA, p : Integer) : PA ==
pp := gcd(v1, v2, p)
dv2 := degree(v2)
dpp := degree(pp)
dv2 = dpp =>
v1
dpp = 0 => mul(v1, v2, p)
tmp1 := new((dv2 + 1) pretend NonNegativeInteger, 0)$PA

tmp2 := new((dv2 - dpp + 1) pretend NonNegativeInteger, 0)$PA

copy_first(tmp1, v2, dv2 + 1)
divide!(tmp1, pp, tmp2, p)
mul(v1, tmp2, p)

lcm(a : PrimitiveArray PA, lo : Integer, hi: Integer, p: Integer) ==
res := a(lo)
for i in (lo + 1)..hi repeat
    res := lcm2(a(i), res, p)
res

inner_mul : (PA, PA, PA, SingleInteger, SingleInteger, SingleInteger, Integer) -> Void

mul(x : PA, y : PA, p : Integer) : PA ==
xdeg := degree(x) pretend SingleInteger
ydeg := degree(y) pretend SingleInteger
if xdeg > ydeg then
tmpp := x
tmp := xdeg
x := y
xdeg := ydeg
y := tmpp
ydeg := tmp
xccoeffs := x
yccoeffs := y
xdeg < 0 => x
inner_mul(x, y, z, xdeg, ydeg, zdeg, p) ==
    if ydeg < xdeg then
        tmpp := x
        tmp := xdeg
        x := y
        xdeg := ydeg
        y := tmpp
        ydeg := tmp
    xdeg :=
        zdeg < xdeg => zdeg
        xdeg
    ydeg :=
        zdeg < ydeg => zdeg
        ydeg
    ss : Integer
    i : SingleInteger
    j : SingleInteger
    for i in 0..xdeg repeat
        ss := z(i)
        for j in 0..i repeat
            ss := Qmuladd(x(i - j), y(j), ss)
        z(i) := Qrem(ss, p)
    for i in (xdeg+1)..ydeg repeat
        ss := z(i)
        for j in 0..xdeg repeat
            ss := Qmuladd(x(j), y(i-j), ss)
        z(i) := Qrem(ss, p)
    for i in (ydeg+1)..zdeg repeat
        ss := z(i)
        for j in (i-xdeg)..ydeg repeat
            ss := Qmuladd(x(i - j), y(j), ss)
        z(i) := Qrem(ss, p)

truncated_mul_add(x, y, z, m, p) ==
    xdeg := (#x - 1) pretend SingleInteger
    ydeg := (#y - 1) pretend SingleInteger
    inner_mul(x, y, z, xdeg, ydeg, m pretend SingleInteger, p)

truncated_multiplication(x, y, m, p) ==
    xdeg := (#x - 1) pretend SingleInteger
    ydeg := (#y - 1) pretend SingleInteger
    z := new((m pretend SingleInteger + 1)
        pretend NonNegativeInteger, 0)$PA
inner_mul(x, y, z, xdeg, ydeg, m pretend SingleInteger, p)

z

pow(x : PA, n : PositiveInteger, d: NonNegativeInteger, _
p : Integer) : PA ==
one? n => x
odd?(n)$Integer =>
truncated_multiplication(x,
pow(truncated_multiplication(x, x, d, p),
shift(n,-1) pretend PositiveInteger,
d,
p),
d,
p)
pow(truncated_multiplication(x, x, d, p),
shift(n,-1) pretend PositiveInteger,
d,
p)

differentiate(x: PA, p: Integer): PA ==
d := #x - 1
if zero? d then empty()$PA
else
  r := new(d::NonNegativeInteger, 0)$PA
  for i in 0..d-1 repeat
    i1 := i+1
    r.i := Qmul(i1, x.i1, p)
  r

differentiate(x: PA, n: NonNegativeInteger, p: Integer): PA ==
zero? n => x
d := #x - 1
if d < n then empty()$PA
else
  r := new((d-n+1) pretend NonNegativeInteger, 0)$PA
  for i in n..d repeat
    j := i-n
    f := j+1
    for k in j+2..i repeat
      f := Qmul(f, k, p)
    r.(j pretend NonNegativeInteger) := Qmul(f, x.i, p)
  r

extended_gcd(x : PA, y : PA, p : Integer) : List(PA) ==
dr0 := degree(x) pretend SingleInteger
dr1 : SingleInteger
swapped : Boolean := false
t0 : PA
if dr0 < 0 then
  (x, y) := (y, x)
  dr1 := dr0
dr0 := degree(x) pretend SingleInteger
swapped := true
else
dr1 := degree(y) pretend SingleInteger
dr1 < 0 =>
  dr0 < 0 =>
    return [new(1, 0)$PA, new(1, 0)$PA, new(1, 1)$PA]
r0 := new((dr0 + 1) pretend NonNegativeInteger, 0)$PA
  copy_first(r0, x, dr0 + 1)
c := r0(dr0)
c := modInverse(c, p)
mul_by_scalar(r0, dr0, c, p)
t0 := new(1, c)$PA
if swapped then
  return [r0, new(1, 0)$PA, t0]
else
  return [r0, t0, new(1, 0)$PA]
swapped => error "impossible"
dt := (dr0 > 0 => dr0 - 1 ; 0)
ds := (dr1 > 0 => dr1 - 1 ; 0)
-- invariant: r0 = s0*x + t0*y, r1 = s1*x + t1*y
r0 := new((dr0 + 1) pretend NonNegativeInteger, 0)$PA
t0 := new((dt + 1) pretend NonNegativeInteger, 0)$PA
s0 := new((ds + 1) pretend NonNegativeInteger, 0)$PA
  copy_first(r0, x, dr0 + 1)
s0(0) := 1
r1 := new((dr1 + 1) pretend NonNegativeInteger, 0)$PA
t1 := new((dt + 1) pretend NonNegativeInteger, 0)$PA
s1 := new((ds + 1) pretend NonNegativeInteger, 0)$PA
  copy_first(r1, y, dr1 + 1)
t1(0) := 1
while dr1 > 0 repeat
  while dr0 >= dr1 repeat
    delta := dr0 - dr1
c1 := sub_SI(p, r0(dr0))$Lisp
c0 := r1(dr1)
    if c0 != 1 and delta > 30 then
      c0 := modInverse(c0, p)
mul_by_scalar(r1, dr1, c0, p)
mul_by_scalar(t1, dt, c0, p)
mul_by_scalar(s1, ds, c0, p)
c0 := 1
r0(dr0) := 0
  dr0 := dr0 - 1
  vector_combination(r0, c0, r1, c1, dr0, delta, p)
  vector_combination(t0, c0, t1, c1, dt, delta, p)
  vector_combination(s0, c0, s1, c1, ds, delta, p)
while r0(dr0) = 0 repeat
  dr0 := dr0 - 1
  if dr0 < 0 then break
\[(r_0, r_1) := (r_1, r_0)\]
\[(d_{r_0}, d_{r_1}) := (d_{r_1}, d_{r_0})\]
\[(s_0, s_1) := (s_1, s_0)\]
\[(t_0, t_1) := (t_1, t_0)\]
\[d_{r_1} \geq 0 \Rightarrow \]
\[c := r_1(0)\]
\[c := \text{modInverse}(c, p)\]
\[r_1(0) := 1\]
\[\text{mul_by_scalar}(s_1, d_s, c, p)\]
\[\text{mul_by_scalar}(t_1, d_t, c, p)\]
\[\text{return } [r_1, s_1, t_1]\]
\[c := r_0(d_{r_0})\]
\[c := \text{modInverse}(c, p)\]
\[\text{mul_by_scalar}(r_0, d_{r_0}, c, p)\]
\[\text{mul_by_scalar}(s_0, d_s, c, p)\]
\[\text{mul_by_scalar}(t_0, d_t, c, p)\]
\[[r_0, s_0, t_0]\]

\[
\text{resultant}(x : \text{PA}, y : \text{PA}, p : \text{Integer}) : \text{Integer} ==
\]
\[d_{r_0} := \text{degree}(x) \text{ pretend } \text{SingleInteger}\]
\[d_{r_0} < 0 \Rightarrow 0\]
\[d_{r_1} := \text{degree}(y) \text{ pretend } \text{SingleInteger}\]
\[d_{r_1} < 0 \Rightarrow 0\]
\[r_0 := \text{new}((d_{r_0} + 1) \text{ pretend } \text{NonNegativeInteger}, 0)\]$\text{PA}$
\[r_0 := \text{new}((d_{r_1} + 1) \text{ pretend } \text{NonNegativeInteger}, 0)\]$\text{PA}$
\[\text{copy_first}(r_0, x, d_{r_0} + 1)\]
\[\text{copy_first}(r_1, y, d_{r_1} + 1)\]
\[\text{res} : \text{SingleInteger} := 1\]
\[\text{repeat}\]
\[d_{r_0} < d_{r_1} \Rightarrow\]
\[(r_0, r_1) := (r_1, r_0)\]
\[(d_{r_0}, d_{r_1}) := (d_{r_1}, d_{r_0})\]
\[c_0 := r_1(d_{r_1})\]
\[d_{r_1} = 0 \Rightarrow\]
\[\text{while } 0 < d_{r_0} \text{ repeat}\]
\[\text{res} := \text{Qmul}(\text{res}, c_0, p)\]
\[d_{r_0} := d_{r_0} - 1\]
\[\text{return } \text{res}\]
\[\text{delta} := d_{r_0} - d_{r_1}\]
\[c_1 := \text{sub_SI}(p, r_0(d_{r_0}))\]$\text{Lisp}$
\[\text{if } c_0 = 1 \text{ then}\]
\[c_1 := \text{Qmul}(c_1, \text{modInverse}(c_0, p), p)\]
\[r_0(d_{r_0}) := 0\]
\[d_{r_0} := d_{r_0} - 1\]
\[\text{vector_combination}(r_0, 1, r_1, c_1, d_{r_0}, \text{delta}, p)\]
\[\text{res} := \text{Qmul}(\text{res}, c_0, p)\]
\[\text{while } r_0(d_{r_0}) = 0 \text{ repeat}\]
\[d_{r_0} := d_{r_0} - 1\]
\[d_{r_0} < 0 \Rightarrow \text{return } 0\]
\[\text{res} := \text{Qmul}(\text{res}, c_0, p)\]
| POLYVEC.dotabb |
| "POLYVEC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYVEC"] |
| "A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"] |
| "POLYVEC" -> "A1AGG" |
Chapter 23

Chapter V

package VECTOR2 VectorFunctions2

— VectorFunctions2.input —

)set break resume
)sys rm -f VectorFunctions2.output
)spool VectorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show VectorFunctions2
--E 1

)spool
)lisp (bye)

— VectorFunctions2.help —

====================================================================
VectorFunctions2 examples
====================================================================

This package provides operations which all take as arguments vectors of elements of some type A and functions from A to another of type B. The operations all iterate over their vector argument and either return a value of type B or a vector over B.
VectorFunctions2 (VECTOR2)

Exports:
map  reduce  scan

— package VECTOR2 VectorFunctions2 —

)abbrev package VECTOR2 VectorFunctions2
++ Description:
++ This package provides operations which all take as arguments
++ vectors of elements of some type \spad{A} and functions from \spad{A} to
++ another of type B. The operations all iterate over their vector argument
++ and either return a value of type B or a vector over B.

VectorFunctions2(A, B):Exports == Implementation where
A, B: Type
VA ==> Vector A
VB ==> Vector B
O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)
UB ==> Union(B,"failed")

Exports ==> with
scan : ((A, B) -> B, VA, B) -> VB
++ scan(func,vec,ident) creates a new vector whose elements are
++ the result of applying reduce to the binary function func,
++ increasing initial subsequences of the vector vec,
reduce : ((A, B) -> B, VA, B) -> B
  ++ reduce(func,vec,ident) combines the elements in vec using the
  ++ binary function func. Argument ident is returned if vec is empty.
map : (A -> B, VA) -> VB
  ++ map(f, v) applies the function f to every element of the vector v
  ++ producing a new vector containing the values.
map : (A -> UB, VA) -> Union(VB,"failed")
  ++ map(f, v) applies the function f to every element of the vector v
  ++ producing a new vector containing the values or \spad{"failed"}.

Implementation ==> add
  scan(f, v, b)  == scan(f, v, b)$O2
  reduce(f, v, b) == reduce(f, v, b)$O2
  map(f:(A->B), v:VA):VB == map(f, v)$O2
  map(f:(A -> UB), a:VA):Union(VB,"failed") ==
    res : List B := []
    for u in entries(a) repeat
      r := f u
      r = "failed" => return "failed"
      res := [r::B,:res]
    vector reverse! res

package VIEWDEF ViewDefaultsPackage

— ViewDefaultsPackage.input —

)set break resume
)sys rm -f ViewDefaultsPackage.output
)spool ViewDefaultsPackage.output
)set message test on
)set message auto off
)clear all
ViewportDefaultsPackage describes default and user definable values for graphics.

See Also:
- `)show View Defaults Package`

### ViewDefaultsPackage (VIEWDEF)

**Exports:**
- axesColorDefault
- pointColorDefault
- lineColorDefault
- pointSizeDefault
- tubePointsDefault
- tubeRadiusDefault
- unitsColorDefault
- var1StepsDefault
- var2StepsDefault
- unitsColorDefault
- viewDefautls
- viewPosDefault
- viewSizeDefault
- viewWriteAvailable
- viewWriteDefault
-- package VIEWDEF ViewDefaultsPackage --

)abbrev package VIEWDEF ViewDefaultsPackage
++ Author: Jim Wen
++ Date Created: 15 January 1990
++ Date Last Updated:
++ Description:
++ ViewportDefaultsPackage describes default and user definable
++ values for graphics

ViewDefaultsPackage():Exports == Implementation where
  I ==> Integer
  C ==> Color
  PAL ==> Palette
  L ==> List
  S ==> String
  E ==> Expression
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  SF ==> DoubleFloat
  B ==> Boolean

writeAvailable => (["PIXMAP","BITMAP","POSTSCRIPT","IMAGE"]::L S)
-- need not worry about case of letters

Exports => with

  pointColorDefault : () -> PAL
  ++ pointColorDefault() returns the default color of points in a 2D
  ++ viewport.

  pointColorDefault : PAL -> PAL
  ++ pointColorDefault(p) sets the default color of points in a 2D viewport
  ++ to the palette p.

  lineColorDefault : () -> PAL
  ++ lineColorDefault() returns the default color of lines connecting
  ++ points in a 2D viewport.

  lineColorDefault : PAL -> PAL
  ++ lineColorDefault(p) sets the default color of lines connecting points
  ++ in a 2D viewport to the palette p.

  axesColorDefault : () -> PAL
  ++ axesColorDefault() returns the default color of the axes in a
  ++ 2D viewport.

  axesColorDefault : PAL -> PAL
  ++ axesColorDefault(p) sets the default color of the axes in a 2D
++ viewport to the palette p.

unitsColorDefault : () -> PAL
++ unitsColorDefault() returns the default color of the unit ticks in
++ a 2D viewport.

unitsColorDefault : PAL -> PAL
++ unitsColorDefault(p) sets the default color of the unit ticks in
++ a 2D viewport to the palette p.

pointSizeDefault : () -> PI
++ pointSizeDefault() returns the default size of the points in
++ a 2D viewport.

pointSizeDefault : PI -> PI
++ pointSizeDefault(i) sets the default size of the points in a 2D
++ viewport to i.

viewPosDefault : () -> L NNI
++ viewPosDefault() returns the default x and y position of a
++ viewport window unless overriden explicitly, newly created
++ viewports will have this x and y coordinate.

viewPosDefault : L NNI -> L NNI
++ viewPosDefault([x,y]) sets the default x and y position of a
++ viewport window unless overriden explicitly, newly created
++ viewports will have the x and y coordinates x, y.

viewSizeDefault : () -> L PI
++ viewSizeDefault() returns the default viewport width and height.

viewSizeDefault : L PI -> L PI
++ viewSizeDefault([w,h]) sets the default viewport width to w and height
++ to h.

viewDefaults : () -> Void
++ viewDefaults() resets all the default graphics settings.

viewWriteDefault : () -> L S
++ viewWriteDefault() returns the list of things to write in a viewport
++ data file; a viewalone file is always generated.

viewWriteDefault : L S -> L S
++ viewWriteDefault(l) sets the default list of things to write in a
++ viewport data file to the strings in l; a viewalone file is always
++ generated.

viewWriteAvailable : () -> L S
++ viewWriteAvailable() returns a list of available methods for writing,
++ such as BITMAP, POSTSCRIPT, etc.
var1StepsDefault : () -> PI
++ var1StepsDefault() is the current setting for the number of steps to
++ take when creating a 3D mesh in the direction of the first defined
++ free variable (a free variable is considered defined when its
++ range is specified (e.g. x=0..10)).

var2StepsDefault : () -> PI
++ var2StepsDefault() is the current setting for the number of steps to
++ take when creating a 3D mesh in the direction of the first defined
++ free variable (a free variable is considered defined when its
++ range is specified (e.g. x=0..10)).

var1StepsDefault : PI -> PI
++ var1StepsDefault(i) sets the number of steps to take when creating a
++ 3D mesh in the direction of the first defined free variable to i
++ (a free variable is considered defined when its range is specified
++ (e.g. x=0..10)).

var2StepsDefault : PI -> PI
++ var2StepsDefault(i) sets the number of steps to take when creating a
++ 3D mesh in the direction of the first defined free variable to i
++ (a free variable is considered defined when its range is specified
++ (e.g. x=0..10)).

tubePointsDefault : PI -> PI
++ tubePointsDefault(i) sets the number of points to use when creating
++ the circle to be used in creating a 3D tube plot to i.

tubePointsDefault : () -> PI
++ tubePointsDefault() returns the number of points to be used when
++ creating the circle to be used in creating a 3D tube plot.

tubeRadiusDefault : Float -> SF -- current tube.spad asks for SF
++ tubeRadiusDefault(r) sets the default radius for a 3D tube plot to r.

tubeRadiusDefault : () -> SF
++ tubeRadiusDefault() returns the radius used for a 3D tube plot.

Implementation ==> add

import Color()
import Palette()
--import StringManipulations()

defaultPointColor : Reference(PAL) := ref bright red()
defaultLineColor : Reference(PAL) := ref pastel green() --bright blue()
defaultAxesColor : Reference(PAL) := ref dim red()
defaultUnitsColor : Reference(PAL) := ref dim yellow()
defaultPointSize : Reference(PI) := ref(3::PI)
defaultXPos : Reference(NNI) := ref(0::NNI)
defaultYPos : Reference(NNI) := ref(0::NNI)
defaultWidth : Reference(PI) := ref(400::PI)
defaultHeight : Reference(PI) := ref(400::PI)
defaultThingsToWrite : Reference(L S) := ref([]::L S)
defaultVar1Steps : Reference(PI) := ref(27::PI)
defaultVar2Steps : Reference(PI) := ref(27::PI)
defaultTubePoints : Reference(PI) := ref(6::PI)
defaultTubeRadius : Reference(SF) := ref(convert(0.5)@SF)
defaultClosed : Reference(B) := ref(false)

--%Viewport window dimensions specifications
viewPosDefault == [defaultXPos(),defaultYPos()]
viewPosDefault l ==
  #l < 2 => error "viewPosDefault expects a list with two elements"
  [defaultXPos() := first l,defaultYPos() := last l]
viewSizeDefault == [defaultWidth(),defaultHeight()]
viewSizeDefault l ==
  #l < 2 => error "viewSizeDefault expects a list with two elements"
  [defaultWidth() := first l,defaultHeight() := last l]
viewDefaults ==
defaultPointColor : Reference(PAL) := ref bright red()
defaultLineColor : Reference(PAL) := ref pastel green() --bright blue()
defaultAxesColor : Reference(PAL) := ref dim red()
defaultUnitsColor : Reference(PAL) := ref dim yellow()
defaultPointSize : Reference(PI) := ref(3::PI)
defaultXPos : Reference(NNI) := ref(0::NNI)
defaultYPos : Reference(NNI) := ref(0::NNI)
defaultWidth : Reference(PI) := ref(400::PI)
defaultHeight : Reference(PI) := ref(427::PI)

--%2D graphical output specifications
pointColorDefault == defaultPointColor()
pointColorDefault p == defaultPointColor() := p
lineColorDefault == defaultLineColor()
lineColorDefault p == defaultLineColor() := p
axesColorDefault == defaultAxesColor()
axesColorDefault p == defaultAxesColor() := p
unitsColorDefault == defaultUnitsColor()
unitsColorDefault p == defaultUnitsColor() := p
pointSizeDefault == defaultPointSize()
pointSizeDefault x == defaultPointSize() := x
-- %3D specific stuff
var1StepsDefault == defaultVar1Steps()
var1StepsDefault i == defaultVar1Steps() := i

var2StepsDefault == defaultVar2Steps()
var2StepsDefault i == defaultVar2Steps() := i

tubePointsDefault == defaultTubePoints()
tubePointsDefault i == defaultTubePoints() := i

tubeRadiusDefault == defaultTubeRadius()
tubeRadiusDefault f == defaultTubeRadius() := convert(f)@SF

-- %File output stuff
viewWriteAvailable == writeAvailable
viewWriteDefault == defaultThingsToWrite()

viewWriteDefault listOfThings ==
thingsToWrite : L S := []
for aTypeOfFile in listOfThings repeat
  if (writeTypeInt := position(upperCase aTypeOfFile, viewWriteAvailable())) < 0 then
    sayBrightly([" > ", concat(aTypeOfFile, " is not a valid file type for writing a viewport")])$Lisp
  else
    thingsToWrite := append(thingsToWrite, [aTypeOfFile])
  defaultThingsToWrite() := thingsToWrite

package VIEW ViewportPackage

"VIEWDEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEWDEF"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VIEWDEF" -> "STRING"

package VIEW ViewportPackage

)set break resume
)sys rm -f ViewportPackage.output
ViewportPackage (VIEW)

Exports:
  coerce  drawCurves  graphCurves
PACKAGE VIEW ViewportPackage

)abbrev package VIEW ViewportPackage
++ Author: Jim Wen
++ Date Created: 30 April 1989
++ Date Last Updated: 15 June 1990
++ Description:
++ ViewportPackage provides functions for creating GraphImages
++ and TwoDimensionalViewports from lists of lists of points.

ViewportPackage():Exports == Implementation where
DROP  ==> DrawOption
GRIMAGE ==> GraphImage
L     ==> List
P     ==> Point DoubleFloat
PAL   ==> Palette
PI    ==> PositiveInteger
VIEW2D ==> TwoDimensionalViewport

Exports == with

graphCurves : (L L P,PAL,PAL,PI,L DROP) -> GRIMAGE
++ graphCurves([[p0],[p1],...,[pn]],ptColor,lineColor,ptSize,[options])
++ creates a \spad{GraphImage} from the list of lists of points, p0
++ throught pn, using the options specified in the list \spad{options}.
++ The graph point color is specified by \spad{ptColor}, the graph line
++ color is specified by \spad{lineColor}, and the size of the points is
++ specified by \spad{ptSize}.

graphCurves : (L L P,L DROP) -> GRIMAGE
++ graphCurves([[p0],[p1],...,[pn]]) creates a \spad{GraphImage} from
++ the list of lists of points indicated by p0 through pn.

drawCurves : (L L P,PAL,PAL,PI,L DROP) -> VIEW2D
++ drawCurves([[p0],[p1],...,[pn]],ptColor,lineColor,ptSize,[options])
++ creates a \spad{TwoDimensionalViewport} from the list of lists of
++ points, p0 throught pn, using the options specified in the list
++ \spad{options}. The point color is specified by \spad{ptColor}, the
++ line color is specified by \spad{lineColor}, and the point size is
++ specified by \spad{ptSize}.

drawCurves : (L L P,L DROP) -> VIEW2D
++ drawCurves([[p0],[p1],...,[pn]],[options]) creates a
++ \spad{TwoDimensionalViewport} from the list of lists of points,
++ p0 throught pn, using the options specified in the list \spad{options};

coerce : GRIMAGE -> VIEW2D
++ coerce(gi) converts the indicated \spad{GraphImage}, gi, into the
++ \spad{TwoDimensionalViewport} form.
Implementation ==> add

import ViewDefaultsPackage
import DrawOptionFunctions0

--% Functions that return GraphImages

graphCurves(listOfListsOfPoints) ==
  graphCurves(listOfListsOfPoints, pointColorDefault(),_
  lineColorDefault(), pointSizeDefault(),nil())

graphCurves(listOfListsOfPoints,optionsList) ==
  graphCurves(listOfListsOfPoints, pointColorDefault(),_
  lineColorDefault(), pointSizeDefault(),optionsList)

graphCurves(listOfListsOfPoints,ptColor,lineColor,ptSize,optionsList) ==
  len := #listOfListsOfPoints
  listOfPointColors : L PAL := [ptColor for i in 1..len]
  listOfLineColors : L PAL := [lineColor for i in 1..len]
  listOfPointSizes : L PI := [ptSize for i in 1..len]
  makeGraphImage(listOfListsOfPoints,listOfPointColors, _
  listOfLineColors,listOfPointSizes,optionsList)

--% Functions that return Two Dimensional Viewports

drawCurves(listOfListsOfPoints,optionsList) ==
  drawCurves(listOfListsOfPoints,pointColorDefault(),_
  lineColorDefault(),pointSizeDefault(),optionsList)

  v := viewport2D()
  options(v,optList)
  g := graphCurves(ptLists,ptColor,lColor,ptSize,optList)
  putGraph(v,g,1)
  makeViewport2D v

--% Coercions

coerce(graf:GRIMAGE):VIEW2D ==
  if (key graf = 0) then makeGraphImage graf
  v := viewport2D()
  title(v,"VIEW2D")
  -- dimensions(v,viewPosDefault().1,viewPosDefault().2,viewSizeDefault().1,viewSizeDefault())
  putGraph(v,graf,1::PI)
  makeViewport2D v
— VIEW.dotabb —

"VIEW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEW"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"VIEW" -> "FIELD"
"VIEW" -> "RADCAT"
"VIEW" -> "FLAGG"
Chapter 24

Chapter W

package WEIER WeierstrassPreparation

— WeierstrassPreparation.input —

)set break resume
)sys rm -f WeierstrassPreparation.output
)spool WeierstrassPreparation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show WeierstrassPreparation
--E 1

)spool
)lisp (bye)

— WeierstrassPreparation.help —

====================================================================
WeierstrassPreparation examples
====================================================================

This package implements the Weierstrass preparation theorem for multivariate power series.

weierstrass(v,p) where v is a variable, and p is a TaylorSeries(R) in
which the terms of lowest degree $s$ must include $c \cdot v^s$ where $c$ is a constant, $s > 0$, is a list of TaylorSeries coefficients $A[i]$ of the equivalent polynomial


such that $p = A \cdot B$, $B$ being a TaylorSeries of minimum degree 0

See Also:

- )show WeierstrassPreparation
WeierstrassPreparation(R): Defn == Impl where
  R : Field
  VarSet==>Symbol
  SMP ==> Polynomial R
  PS ==> InnerTaylorSeries SMP
  NNI ==> NonNegativeInteger
  ST ==> Stream
  StS ==> Stream SMP
  STPS==>StreamTaylorSeriesOperations
  STTAYLOR==>StreamTaylorSeriesOperations
  SUP==> SparseUnivariatePolynomial(SMP)
  ST2==>StreamFunctions2
  SMPS==> TaylorSeries(R)
  L==>List
  null ==> empty?
  likeUniv ==> univariate
  coef ==> coefficient$SUP
  nil ==> empty

Defn ==> with

crest:(NNI->( StS-> StS))
  ++\spad{crest n} is used internally.
cfirst:(NNI->( StS-> StS))
  ++\spad{cfirst n} is used internally.
sts2stst:(VarSet,StS)->ST StS
  ++\spad{sts2stst(v,s)} is used internally.
clikeUniv:VarSet->(SMP->SUP)
  ++\spad{clikeUniv(v)} is used internally.
weierstrass:(VarSet,SMPS)->L SMPS
  ++\spad{weierstrass(v,ts)} where v is a variable and ts is
  ++ a TaylorSeries, implements the Weierstrass Preparation
  ++ Theorem. The result is a list of TaylorSeries that
  ++ are the coefficients of the equivalent series.
qqq:(NNI,SMPS,ST SMPS)->((ST SMPS)->ST SMPS)
  ++\spad{qqq(n,s,st)} is used internally.

Impl ==> add
import TaylorSeries(R)
import StreamTaylorSeriesOperations SMP
import StreamTaylorSeriesOperations SMPS

map1==>map$(ST2(SMP,SUP))
map2==>map$(ST2(StS,SMP))
map3==>map$(ST2(StS,StS))
transback:ST SMPS->L SMPS
transback smps==
if null smps
then nil()$(L SMPS)
else
  if null first (smps:(ST StS))
  then nil()$(L SMPS)
  else
    cons(map2(first,smps:ST StS):SMPS,
         transback(map3(rest,smps:ST StS):(ST SMPS)))$(L SMPS)

clikeUniv(var)==p +-> likeUniv(p,var)
mind:(NNI,StS)->NNI
mind(n, sts)==
  if null sts
  then error "no mindegree"
  else if first sts=0
    then mind(n+1,rest sts)
    else n
mindegree (sts:StS):NNI== mind(0,sts)

streamlikeUniv:(SUP,NNI)->StS
streamlikeUniv(p:SUP,n:NNI): StS ==
  if n=0
  then cons(coef (p,0),nil()$StS)
  else cons(coef (p,n),streamlikeUniv(p,(n-1):NNI))

transpose:ST StS->ST StS
transpose(s:ST StS)==delay(
  if null s
  then nil()$(ST StS)
  else cons(map2(first,s),transpose(map3(rest,rst s))))

zp==>map$StreamFunctions3(SUP,NNI,StS)
sts2stst(var, sts)==
  zp((x,y) +-> streamlikeUniv(x,y),
     map1(clikeUniv var, sts),(integers 0):(ST NNI))

tp:(VarSet,StS)->ST StS
tp(v,sts)==transpose sts2stst(v,sts)
map4==>map$(ST2 (StS,StS))
maptake:(NNI,ST StS)->ST SMPS
maptake(n,p)== map4(cfirst n,p) pretend ST SMPS
mapdrop:(NNI,ST StS)->ST SMPS
mapdrop(n,p)== map4(crest n,p) pretend ST SMPS
YSS==>Y$ParadoxicalCombinatorsForStreams(SMPS)
weier:(VarSet,StS)->ST SMPS
weier(v,sts)==
  a:=mindegree sts
if \( a = 0 \)
then error "has constant term"
else
\[
p := t v (v, s t s) \text{ pretend } (S T S M P S)
\]
\[
b : S t S := \text{rest}((\text{first } p \text{ pretend } S t S), a :: N N I)
\]
c := retractIfCan first b
c case "failed" => _
error "the coefficient of the lowest degree of the variable should be a constant"
\[
e := \text{recip } b
\]
f := if e case "failed"
then error "no reciprocal"
else e :: S t S
\[
q := (Y S S \text{ qqq}(a, f : S M P S, \text{rest } p))
\]
\[
\text{maptake}(a, (p * q) \text{ pretend } S T S t S)
\]
\[
c f i r s t \ n \ = \ s \ + - > \ \text{first}(s, n) \$ S t S
\]
c r e s t \ n \ = \ s \ + - > \ \text{rest}(s, n) \$ S t S
\[
q q : (N N I, S M P S, S T S M P S, S T S M P S) \to S T S M P S
\]
\[
q q(a, e, p, c) =
\]
\[
\text{cons}(e, (-e) \text{* mapdrop}(a, (p * c) \text{pretend}(S T S)))
\]
\[
q q q(a, e, p) =
\]
\[
s \ + - > \ \text{qq}(a, e, p, s)
\]
\[
\text{we i} : (V a r S e t, S M P S) \to S T S M P S
\]
\[
\text{we i}(v : V a r S e t, s : S M P S) = \text{weier}(v, s : S t S)
\]
\[
\text{we i e r s t r a s s}(v, s m p s) = \text{transback we i}(v, s m p s)
\]
WildFunctionFieldIntegralBasis (WFFINTBS)

Exports:
  integralBasis  localIntegralBasis
— package WFFINTBS WildFunctionFieldIntegralBasis —

)abbrev package WFFINTBS WildFunctionFieldIntegralBasis
++ Authors: Victor Miller, Clifton Williamson
++ Date Created: 24 July 1991
++ Date Last Updated: 20 September 1994
++ Description:
++ In this package K is a finite field, R is a ring of univariate
++ polynomials over K, and F is a framed algebra over R. The package
++ provides a function to compute the integral closure of R in the quotient
++ field of F as well as a function to compute a "local integral basis"
++ at a specific prime.

WildFunctionFieldIntegralBasis(K,R,UP,F): Exports == Implementation where
  K : FiniteFieldCategory
  R : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R
  F : FramedAlgebra(R,UP)

I   ==> Integer
Mat ==> Matrix R
NNI ==> NonNegativeInteger
SAE ==> SimpleAlgebraicExtension
RResult ==> Record(basis: Mat, basisDen: R, basisInv:Mat)
IRResult ==> Record(basis: Mat, basisDen: R, basisInv:Mat,discr: R)
MATSTOR ==> StorageEfficientMatrixOperations

Exports == with
  integralBasis : () -> RResult
    ++ \spar{integralBasis()} returns a record
    ++ \spar{\spar{basis,\spar{basisDen,basisInv}}} containing information regarding
    ++ the integral closure of R in the quotient field of F, where
    ++ F is a framed algebra with R-module basis \spar{w1,w2,...,wn}.
    ++ If \spar{\spar{basis}} is the matrix \spar{\spar{\spar{a_{ij}, i = 1..n, j = 1..n}}}, then
    ++ the \spar{i}th element of the integral basis is
    ++ \spar{\spar{vi = (1/\spar{basisDen}) * \spar{\sum_{j = 1..n} a_{ij} * w_j}}, i.e. the
    ++ \spar{i}th row of \spar{\spar{basis}} contains the coordinates of the
    ++ \spar{i}th basis vector. Similarly, the \spar{i}th row of the
    ++ matrix \spar{\spar{basisInv}} contains the coordinates of \spar{\spar{wi}} with
    ++ respect to the basis \spar{\spar{v1,...,vn}}: if \spar{\spar{basisInv}} is the
    ++ matrix \spar{\spar{\spar{b_{ij}, i = 1..n, j = 1..n}}}, then
    ++ \spar{\spar{wi = \spar{\sum_{j = 1..n} b_{ij} * v_j}}, j = 1..n}.
localIntegralBasis : R -> RResult
  ++ \spar{localIntegralBasis(p)} returns a record
  ++ \spar{\spar{basis,\spar{basisDen,basisInv}}} containing information regarding
  ++ the local integral closure of R at the prime \spar{p} in the quotient
  ++ field of F, where F is a framed algebra with R-module basis
  ++ \spar{\spar{w1,w2,...,wn}}.
++ If \( \text{basis} \) is the matrix \( \{a_{ij}, \ i = 1..n, \ j = 1..n\} \), then
++ the \( \text{basis}[i] \)th element of the local integral basis is
++ \( \text{basis}[v_i] = (1/\text{basisDen}) \times \sum(a_{ij} \times w_j, \ j = 1..n) \), i.e. the
++ \( \text{basis}[i] \)th row of \( \text{basis} \) contains the coordinates of the
++ \( \text{basis}[i] \)th basis vector. Similarly, the \( \text{basis}[i] \)th row of the
++ matrix \( \text{basis}[\text{basisInv}] \) contains the coordinates of \( \text{basis}[w_i] \) with
++ respect to the basis \( \text{basis}[v_1,...,v_n] \): if \( \text{basis}[\text{basisInv}] \) is the
++ matrix \( \text{basis}[\{b_{ij}, \ i = 1..n, \ j = 1..n\}] \), then
++ \( \text{basis}[w_i] = \sum(b_{ij} \times v_j, \ j = 1..n) \).

Implementation ==> add
import IntegralBasisTools(R, UP, F)
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)
import DistinctDegreeFactorize(K,R)

listSquaredFactors: R -> List R
listSquaredFactors px ==
-- returns a list of the factors of px which occur with
-- exponent > 1
ans : List R := empty()
factored := factor(px)$DistinctDegreeFactorize(K,R)
for f in factors(factored) repeat
  if f.exponent > 1 then ans := concat(f.factor,ans)
ans

iLocalIntegralBasis: (Vector F,Vector F,Matrix R,Matrix R,R,R) -> IResult
iLocalIntegralBasis(bas,pows,tfm,matout,dis,pr) ==
  n := rank(F)$F; standardBasis := basis(F)$F
  -- 'standardBasis' is the basis for F as a FramedAlgebra;
  -- usually this is \[1, y, y^2, ..., y^{(n-1)}\]
  p2 := pr * pr; sae := SAE(K,R,pr)
  p := characteristic(F)$F; q := size(sae)$F
  lp := leastPower(q,n)
  rb := scalarMatrix(n,1); rbinv := scalarMatrix(n,1)
  -- rb = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  rbden : R := 1; index : R := 1; oldIndex : R := 1
  repeat
    -- pows = \[(w_1 \times rbden)^q, ..., (w_n \times rbden)^q\], where
    -- bas = \[w_1, ..., wn\] is 'rbden' times the basis for the order B = 'rb'
    for i in 1..n repeat
      bi : F := 0
      for j in 1..n repeat
        bi := bi + qelt(rb,i,j) * qelt(standardBasis,j)
      qsetelt!(bas,i,bi)
    qsetelt!(pows,bi ** p)
coor0 := transpose coordinates(pows,bas)
denPow := rbden ** ((p - 1) :: NNI)
(coMat0 := coor0 exquo denPow) case "failed" =>
   error "can't happen"
-- the jth column of coMat contains the coordinates of (wj/rbden)**q
-- with respect to the basis [w1/rbden,...,wn/rbden]
coMat := coMat0 :: Matrix R
-- the ith column of 'pPows' contains the coordinates of the pth power
-- of the ith basis element for B/prime.B over 'sae' = R/prime.R
pPows := map(reduce,coMat)$MatrixCategoryFunctions2(R,Vector R,
   Vector R,Vector sae,Vector sae,Vector sae,Matrix sae)
-- 'frob' will eventually be the Frobenius matrix for B/prime.B over
-- 'sae' = R/prime.R; at each stage of the loop the ith column will
-- contain the coordinates of p^k-th powers of the ith basis element
frob := copy pPows; tmpMat : Matrix sae := new(n,n,0)
for r in 2..leastPower(p,q) repeat
   for i in 1..n repeat for j in 1..n repeat
      qsetelt_!(tmpMat,i,j,qelt(frob,i,j) ** p)
      times_!(frob,pPows,tmpMat)$MATSTOR(sae)
   }
   frobPow := frob ** lp
   -- compute the p-radical
   ns := nullSpace frobPow
   for i in 1..n repeat for j in 1..n repeat qsetelt_!(tfm,i,j,0)
   for vec in ns for i in 1..repeat
      for j in 1..n repeat
         qsetelt_!(tfm,i,j,lift qelt(vec,j))
   } id := squareTop rowEchelon(tfm,prime)
-- id = basis matrix of the p-radical
idinv := UpTriBddDenomInv(id, prime)
-- id * idinv = prime * identity
-- no need to check for inseparability in this case
rbinv := idealiser(id * rb, rbinv * idinv, prime * rbden)
index := diagonalProduct rbinv
rb := rowEchelon LowTriBddDenomInv(rbinv,rbden * prime)
if divideIfCan_!(rb,matrixOut,prime,n) = 1
   then rb := matrixOut
else rbden := rbden * prime
rbinv := UpTriBddDenomInv(rb,rbden)
indexChange := index quo oldIndex
oldIndex := index
disc := disc quo (indexChange * indexChange)
(not sizeLess?(1,indexChange)) or ((disc exquo p2) case "failed") =>
   return [rb, rbden, rbinv, disc]
integralBasis() ==
   traceMat := traceMatrix()$F; n := rank()$F
disc := determinant traceMat
-- discriminant of current order
zero? disc => error "integralBasis: polynomial must be separable"
singList := listSquaredFactors disc -- singularities of relative Spec
runningRb := scalarMatrix(n,1); runningRbinv := scalarMatrix(n,1)
-- runningRb = basis matrix of current order
-- runningRbinv = inverse basis matrix of current order
-- these are w.r.t. the original basis for F
runningRbden : R := 1
-- runningRbden = denominator for current basis matrix
empty? singList => [runningRb, runningRbden, runningRbinv]
bas : Vector F := new(n,0); pows : Vector F := new(n,0)
-- storage for basis elements and their powers
tfm : Matrix R := new(n,n,0)
-- 'tfm' will contain the coordinates of a lifting of the kernel
-- of a power of Frobenius
matrixOut : Matrix R := new(n,n,0)
for prime in singList repeat
  lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)
  rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
disc := lb.dscr
-- update 'running integral basis' if newly computed
-- local integral basis is non-trivial
if sizeLess?(1,rbden) then
  mat := vertConcat(rbden * runningRb,runningRbden * rb)
  runningRbden := runningRbden * rbden
  runningRb := squareTop rowEchelon(mat,runningRbden)
  runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
[runningRb, runningRbden, runningRbinv]

localIntegralBasis prime ==
  traceMat := traceMatrix()$F; n := rank()$F
disc := determinant traceMat -- discriminant of current order
zero? disc => error "localIntegralBasis: polynomial must be separable"
(disc exquo (prime * prime)) case "failed" =>
  [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
bas : Vector F := new(n,0); pows : Vector F := new(n,0)
-- storage for basis elements and their powers
tfm : Matrix R := new(n,n,0)
-- 'tfm' will contain the coordinates of a lifting of the kernel
-- of a power of Frobenius
matrixOut : Matrix R := new(n,n,0)
lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)
[lb.basis, lb.basisDen, lb.basisInv]
Chapter 25

Chapter X

package XEXPPKG XExponentialPackage

— XExponentialPackage.input —

)set break resume
)sys rm -f XExponentialPackage.output
)spool XExponentialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show XExponentialPackage
--E 1

)spool
)lisp (bye)

— XExponentialPackage.help —

====================================================================
XExponentialPackage examples
====================================================================

This package provides computations of logarithms and exponentials for polynomials in non-commutative variables.

See Also:

3743
XExponentialPackage (XEXPPKG)

Exports:

exp  Hausdorff  log

— XEXPPKG.dotabb —

"XEXPPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=XEXPPKG"]

— package XEXPPKG XExponentialPackage —

)abbrev package XEXPPKG XExponentialPackage
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This package provides computations of logarithms and exponentials
++ for polynomials in non-commutative variables.

XExponentialPackage(R, VarSet, XPOLY): Public == Private where

RN  ==> Fraction Integer
NNI ==> NonNegativeInteger
I   ==> Integer
R   : Join(Ring, Module RN)
-- R  : Field
VarSet : OrderedSet
XPOLY : XPolynomialsCat(VarSet, R)

Public == with
  exp: (XPOLY, NNI) -> XPOLY
    ++ \texttt{exp(p, n)} returns the exponential of \texttt{p}
    ++ truncated at order \texttt{n}.
  log: (XPOLY, NNI) -> XPOLY
    ++ \texttt{log(p, n)} returns the logarithm of \texttt{p}
    ++ truncated at order \texttt{n}.
  Hausdorff: (XPOLY, XPOLY, NNI) -> XPOLY
    ++ \texttt{Hausdorff(a,b,n)} returns \texttt{log(exp(a)*exp(b))}
    ++ truncated at order \texttt{n}.

Private == add
  log (p,n) ==
    p1 : XPOLY := p - 1
    not quasiRegular? p1 =>
      error "constant term <> 1, impossible log"
    s : XPOLY := 0 -- resultat
    k : I := n :: I
    for i in 1 .. n repeat
      k1 : RN := 1/k
      k2 : R := k1 * 1$R
      s := trunc( trunc(p1,i) * (k2 :: XPOLY - s) , i)
      k := k - 1
    s
  exp (p,n) ==
    not quasiRegular? p =>
      error "constant term <> 0, exp impossible"
    p = 0 => 1
    s : XPOLY := 1$XPOLY -- resultat
    k : I := n :: I
    for i in 1 .. n repeat
      k1 : RN := 1/k
      k2 : R := k1 * 1$R
      s := trunc( 1 +$XPOLY k2 * trunc(p,i) * s , i)
      k := k - 1
    s
  Hausdorff(p,q,n) ==
    p1: XPOLY := exp(p,n)
    q1: XPOLY := exp(q,n)
    log(p1*q1, n)
Chapter 26

Chapter Y
Chapter 27

Chapter Z

package ZDSOLVE ZeroDimensionalSolvePackage

Based on triangular decompositions and the RealClosure constructor, the package ZeroDimensionalSolvePackage provides operations for computing symbolically the real or complex roots of polynomial systems with finitely many solutions.

— ZeroDimensionalSolvePackage.input —

)set break resume
)spool ZeroDimensionalSolvePackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 28
R := Integer
--R
--R
--R (1) Integer
--R
--E 1

--S 2 of 28
ls : List Symbol := [x,y,z,t]
--R
--R
--R (2) [x,y,z,t]
--R
--E 2

--S 3 of 28
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
--R

3749
pack := ZDSOLVE(R, ls, ls2)

p1 := x**2*y*z + x*y**2*z + x*y*z**2 + x*y + x*y + x + y + 1

p2 := x**2*y**2*z + x*y**2*z**2 + x**2*y*z + x*y*z + y + z + 1

p3 := x**2*y**2*z**2 + x**2*y**2*z + x*y**2*z + x*y + x + 1

lp := [p1, p2, p3]
--R x y z + ((x + x)y + x y + x + 1)z + 1]
--R Type: List(Polynomial(Integer))

triangSolve(lp)$pack

(9)

{ 20 19 18 17 16 15 14 13 12
   z - 6z - 41z + 71z + 106z + 92z + 197z + 145z + 257z
   + 11 10 9 8 7 6 5 4 3
   278z + 201z + 278z + 257z + 145z + 197z + 92z + 106z + 71z
   + 2
   - 41z - 6z + 1
   ,
   19 18 17 16
   14745844z + 50357474z - 130948857z - 185261586z
   + 15 14 13 12
   - 18007775z - 338007307z - 275379623z - 453190404z
   + 11 10 9 8
   - 474597456z - 366147695z - 481433567z - 430613166z
   + 7 6 5 4
   - 261878358z - 326073537z - 163008796z - 177213227z
   + 3 2
   - 104356756z + 65241699z + 9237732z - 1567348
   * y
   + 19 18 17 16 15
   1917314z + 6508991z - 16973165z - 24000259z - 23349192z
   + 14 13 12 11 10
   - 43786426z - 35696474z - 58724172z - 61480792z - 47452440z
   + 9 8 7 6 5
   - 62378085z - 55776527z - 33940618z - 42233406z - 21122875z
   + 4 3 2
   - 22958177z - 13504569z + 8448317z + 1195888z - 202934
   ,}
Type: List(RegularChain(Integer,[x,y,z,t]))

univariateSolve(lp)$pack

(10)

complexRoots =

\[
\begin{align*}
&12 \quad 11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \\
\end{align*}
\]

coordinates =

\[
\begin{align*}
&11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \\
&63x + 62%A - 721%A + 1220%A + 705%A - 285%A + 1512%A
\end{align*}
\]

\[
\begin{align*}
&5 \quad 4 \quad 3 \quad 2 \\
&- 735%A + 1401%A - 21%A + 215%A + 1577%A - 142
\end{align*}
\]

\[
\begin{align*}
&11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \\
&63y - 75%A + 890%A - 1682%A - 516%A + 588%A - 1953%A
\end{align*}
\]

\[
\begin{align*}
&5 \quad 4 \quad 3 \quad 2 \\
&1323%A - 1815%A + 426%A - 243%A - 1801%A + 679
\end{align*}
\]

z - %A]

coordinates =

\[
\begin{align*}
&6 \quad 5 \quad 4 \quad 3 \quad 2 \\
&[complexRoots= ? + 5? + 1,coordinates= [x - 1,y - 1,z - %A]]
\end{align*}
\]

Type: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer))))
lr := realSolve(lp)$pack
--R
--R (11)
--R [
--R [%B1,
--R
--R 1184459  19  2335702  18  5460230  17  79900378  16
--R  ------- %B1 - ------- %B1 - ------- %B1 + ------- %B1
--R  1645371  548457  182819  1645371
--R +
--R 43953929  15  13420192  14  553986  13  193381378  12
--R  ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
--R  548457  182819  3731  1645371
--R +
--R 35978916  11  358660781  10  271667666  9  118784873  8
--R  ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
--R  182819  1645371  1645371  548457
--R +
--R 337505020  7  1389370  6  688291  5  3378002  4
--R  ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
--R  1645371  11193  4459  42189
--R +
--R 140671876  3  32325724  2  8270  9741532
--R  ------- %B1 + ------- %B1 - ---- %B1 - -------
--R  1645371  548457  343  1645371
--R ,
--R
--R 91729  19  487915  18  4114333  17  12769878  16
--R = ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
--R  705159  705159  705159  705159
--R +
--R 13243117  15  16292173  14  26536060  13  722714  12
--R = ------- %B1 - ------- %B1 - ------- %B1 - ------- %B1
--R  705159  705159  705159  18081
--R +
--R 5382578  11  15449995  10  14279770  9  6603890  8
--R = ------- %B1 - ------- %B1 - ------- %B1 - ------- %B1
--R  100737  235053  235053  100737
--R +
--R 409930  7  37340389  6  34893715  5  26686318  4
--R = ------- %B1 - ------- %B1 - ------- %B1 - ------- %B1
--R  6027  705159  705159  705159
--R +
--R 801511  3  17206178  2  4406102  377534
--R = ------- %B1 - ------- %B1 - ------- %B1 + ------- %B1
--R  26117  705159  705159  705159
--R ]
--R ,
- ----- %B7 + ----- %B7 + ----- %B7 - ----- %B7
- R 705159 705159 705159 235053
- R +
- R 13243117 15 16292173 14 26536060 13 722714 12
- R - ----- %B7 - ----- %B7 - ----- %B7 - ----- %B7
- R 705159 705159 705159 18081
- R +
- R 5382578 11 15449995 10 14279770 9 6603890 8
- R - ----- %B7 - ----- %B7 - ----- %B7 - ----- %B7
- R 100737 235053 235053 100737
- R +
- R 409930 7 37340389 6 34893715 5 26686318 4
- R - ----- %B7 - ----- %B7 - ----- %B7 - ----- %B7
- R 6027 705159 705159 705159
- R +
- R 801511 3 17206178 2 4406102 377534
- R - ----- %B7 - ----- %B7 - ----- %B7 + ----- %B7
- R 26117 705159 705159 705159

[ %B8,
- R 1184459 19 2335702 18 5460230 17 79900378 16
- R - ----- %B8 - ----- %B8 - ----- %B8 + +----- %B8
- R 1645371 548457 182819 1645371
- R +
- R 43953929 15 13420192 14 553986 13 193381378 12
- R - ----- %B8 + ----- %B8 + ----- %B8 + ----- %B8
- R 548457 182819 3731 1645371
- R +
- R 35979916 11 358660781 10 271667666 9 118784873 8
- R - ----- %B8 + ----- %B8 + ----- %B8 + ----- %B8
- R 182819 1645371 1645371 548457
- R +
- R 337505020 7 1389370 6 688291 5 3378002 4
- R - ----- %B8 + ----- %B8 + ----- %B8 + ----- %B8
- R 1645371 11193 4459 42189
- R +
- R 140671876 3 32325724 2 8270 9741532
- R - ----- %B8 + ----- %B8 - ----- %B8 - ----- %B8
- R 1645371 548457 343 1645371

, ,
- R 91729 19 487915 18 4114333 17 1276987 16
- R - ----- %B8 + ----- %B8 + ----- %B8 - ----- %B8
- R 705159 705159 705159 235053
- R +
- R 13243117 15 16292173 14 26536060 13 722714 12
- R - ----- %B8 - ----- %B8 - ----- %B8 - ----- %B8


(12) Type: List(List(RealClosure(Fraction(Integer))))

(13) Type: PositiveInteger

[ approximate(r,1/1000000) for r in point for point in lr]
--R Type: List(List(Fraction(Integer)))
--E 13

--S 14 of 28
lpr := positiveSolve(lp)$pack
--R
--R
--R (14) []
--R Type: List(List(RealClosure(Fraction(Integer))))
--E 14

--S 15 of 28
f0 := x**3 + y + z + t- 1
--R
--R
--R 3
--R (15) z + y + x + t - 1
--R Type: Polynomial(Integer)
--E 15

--S 16 of 28
f1 := x + y**3 + z + t -1
--R
--R
--R 3
--R (16) z + y + x + t - 1
--R Type: Polynomial(Integer)
--E 16

--S 17 of 28
f2 := x + y + z**3 + t-1
--R
--R
--R 3
--R (17) z + y + x + t - 1
--R Type: Polynomial(Integer)
--E 17

--S 18 of 28
f3 := x + y + z + t**3 -1
--R
--R
--R 3
CHAPTER 27. CHAPTER Z

--R (18) \[ z + y + x + t - 1 \]
--R Type: Polynomial(Integer)
--E 18

--S 19 of 28
lf := \[ f_0, f_1, f_2, f_3 \]
--R
--R
--R (19)
--R \[ 3 3 3 3 \]
--R \[ z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1 \]
--R Type: List(Polynomial(Integer))
--E 19

--S 20 of 28
lts := triangSolve(lf)$pack
--R
--R
--R (20)
--R \[ 2 3 3 \]
--R \[ \{ t + t + 1, z - z - t + t, \]
--R
--R 6 3 2 9 6 3
--R \[ (3t - 6t + 3)z + t - 3t + 5t - 3t \]
--R
--R \[ x + y + z \}
--R
--R
--R 16 13 10 7 4 2
--R \[ \{ t - 6t + 9t + 4t + 15t - 54t + 27, \]
--R
--R 15 14 13 12 11
--R \[ 4907232t + 40893984t - 115013088t + 22805712t + 36330336t \]
--R
--R 10 9 8 7 6
--R \[ 162959040t - 159859440t - 156802608t + 117168768t \]
--R
--R 6 5 4 3
--R \[ 126282384t - 129351600t + 306646992t + 475302816t \]
--R
--R 2
--R \[ - 1006837776t - 237269088t + 480716208 \]
--R
--R \[ z \]
--R
--R 54 51 48 46 45 43 42
\[ 48t - 912t + 8232t - 72t - 46848t + 1152t + 186324t + R 3767 \]
\[ -R 40 + 39 38 37 36 35 \]
\[ -R - 3780t - 543144t - 3168t - 21384t + 1175251t + 41184t + R 278003t - 1843242t - 301815t - 1440726t + 1912012t + R 29 + 28 27 26 25 \]
\[ -R 1442826t + 4696262t - 922481t - 4816188t - 10583524t + R 24 + 23 22 21 20 \]
\[ -R - 208751t + 11472138t + 16762859t - 857663t - 19328175t + R 19 + 18 17 16 15 \]
\[ -R - 18270421t + 4914903t + 22483044t + 12926517t - 8605511t + R 14 + 13 12 11 10 \]
\[ -R - 17455518t - 5014597t + 8108814t + 9465535t + 190542t + R 9 + 8 7 6 5 4 \]
\[ -R - 4305624t - 226123t + 661905t + 1169775t + 226260t - 209952t + R + 3 \]
\[ -R - 141183t + 27216t + 3 \]
\[ -R \]
\[ 3 2 2 3 6 3 3 2 \]
\[ (3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z + R 6 + 3 \]
\[ 9 6 3 \]
\[ (3t - 6t + 3)z + t - 3t + 5t - 3t + R , 3 \]
\[ x + y + z + t - 1} \]
\[ , 2 \]
\[ \{t, z - 1, y - 1, x + y}, \{t - 1, z, y - 1, x + y}, \{t - 1, z - 1, y + 1, x\}, \]
\[ 16 13 10 7 4 2 \]
\[ \{t - 6t + 9t + 4t + 15t - 54t + 27, \]
\[ 29 + 28 27 26 25 \]
\[ -R 4907232t + 40893984t - 115013088t - 1730448t - 168139584t + R 24 + 23 22 21 \]
\[ -R 738024880t - 195372288t + 315849456t + 2567279232t + R + 20 + 19 18 17 \]
\[ -R 937147968t + 1026357696t + 4780488240t - 2893767696t \]
\[ (3z + (6t - 6)z + (6t - 12t + 3)z + 2t - 6t + t + 3t)y + (3t - 3)z + (6t - 12t + 6)z + (4t - 12t + 11t - 3)z + t - 4t + 5t - 2t + 3x + y + z + t - 1 \]

\[ t - 1, z - 1, y, x + z \}

\[ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \]

\[ t + t + t - 2t - 2t - 2t + 19t + 19t - 8 \]

\[ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \]

\[ 2395770t + 3934440t - 3902067t - 10084164t - 1010448t \]

\[ 32386932t + 22413225t - 10432368 \]

\[ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \]

\[ - 463519t + 3586833t + 9494955t - 8539305t - 33283098t \]

\[ 2 \]

\[ 35479377t + 46263256t - 17419896 \]

\[ 3 \ 4 \ 6 \ 3 \ 3 \ 3 \ 2 \ 3 \]

\[ 3z + (9t - 9)z + (12t - 24t + 9)z + (- 152t + 219t - 67)z + 6 \ 4 \ 3 \]

\[ - 41t + 57t + 25t - 57t + 16 \]

\[ * \]

\[ 4 \ 3 \ 3 \ 6 \ 3 \ 2 \ 3 \]

\[ 3z + (9t - 9)z + (12t - 24t + 9)z + (- 152t + 219t - 67)z \]

\[ 6 \ 4 \ 3 \]

\[ - 41t + 57t + 25t - 57t + 16 \]

\[ * \]

\[ y \]

\[ 3 \ 4 \ 6 \ 3 \ 3 \ 3 \ 2 \ 3 \]

\[ (3t - 3)z + (9t - 18t + 9)z + (- 181t + 270t - 89)z \]
---R +
---R 6 4 3
---R (- 92t + 135t + 49t - 135t + 43)z + 27t - 27t - 54t + 396t
---R +
---R - 486t + 144
---R ,
---R 3
---R x + y + z + t - 1} 
---R ,
---R 3
---R {t, z - t + 1, y - 1, x - 1}, {t - 1, z, y, x}, {t, z - 1, y, x}, {t, z, y - 1, x}, {t, z, y, x - 1}
---R Type: List(RegularChain(Integer,[x,y,z,t]))
---E 20

---S 21 of 28
univariateSolve(lf)$pack
---R
---R (21)
---R [[complexRoots= ?,coordinates= [x - 1, y - 1, z + 1, t - %A]],
---R [complexRoots= ?,coordinates= [x, y - 1, z, t - %A]],
---R [complexRoots= ? - 1,coordinates= [x, y, z, t - %A]],
---R [complexRoots= ?,coordinates= [x - 1, y, z, t - %A]],
---R [complexRoots= ?,coordinates= [x, y, z - 1, t - %A]],
---R [complexRoots= ?,coordinates= [x - 1, y, z - 1, t]],
---R [complexRoots= ? - 1,coordinates= [x - 1, y + 1, z - 1, t]],
---R [complexRoots= ? + 1,coordinates= [x - 1, y - 1, z, t - %A]]
---R
---R 6 3 2
---R [complexRoots= ? - 2? + 3? - 3, 
---R 3 3
---R coordinates= [2x + %A + %A - 1, 2y + %A + %A - 1, z - %A, t - %A]]
---R ,
---R
---R 5 3 2
---R [complexRoots= ? + 3? - 2? + 3? - 3, 
---R 3
---R coordinates= [x - %A, y - %A, z + %A + 2%A - 1, t - %A]]
---R ,
---R
---R 4 3 2
---R [complexRoots= ? - ? - 2? + 3, 
---R 3 3
---R coordinates= [x + %A - %A - 1, y + %A - %A - 1, z - %A + 2%A + 1, t - %A]]
---R ,
---R [complexRoots= ? + 1,coordinates= [x - 1, y - 1, z, t - %A]],
---R
---R 6 3 2
--R [complexRoots= ? + 2? + 3? - 3,
--R 3
--R coordinates= [2x - %A - %A - 1,y + %A,2z - %A - %A - 1,t + %A]]
--R ,
--R
--R 6 4 3 2
--R coordinates =
--R 5 4 3 2
--R [12609x + 23%A + 49%A - 46%A + 362%A - 5015%A - 8239,
--R 5 4 3 2
--R 25218y + 23%A + 49%A - 46%A + 362%A + 7594%A - 8239,
--R 5 4 3 2
--R 25218z + 23%A + 49%A - 46%A + 362%A + 7594%A - 8239,
--R 5 4 3 2
--R 12609t + 23%A + 49%A - 46%A + 362%A - 5015%A - 8239]
--R ]
--R ,
--R
--R 5 3 2
--R 3
--R coordinates= [8x + %A + 8%A - 8,2y - %A,2z - %A,2t - %A]]
--R ,
--R
--R 5 4 3 2
--R coordinates =
--R 3 3 3
--R [2x - %A + 2%A - 1, 2y + %A - 4%A + 1, 2z - %A + 2%A - 1,
--R 3
--R 2t - %A + 2%A - 1]
--R ]
--R ,
--R
--R 4 3 2
--R [complexRoots= ? - 3? + 4? - 6? + 13,
--R coordinates =
--R 3 2 3
--R [9x - 2%A + 4%A - %A + 2, 9y + %A - 2%A + 5%A - 1,
--R 3 2 3
--R 9z + %A - 2%A + 5%A - 1, 9t + %A - 2%A - 4%A - 1]
--R ]
--R ,
--R
--R 4 2
--R [complexRoots= ? - 11? + 37,
```plaintext
coordinates = 2 2 2 2
[3x + %A + 7,6y + %A + 3%A - 7,3z - %A + 7,6t + %A - 3%A - 7]

, [complexRoots= ? + 1,coordinates= [x - 1,y,z - 1,t + 1]],
[complexRoots= ? + 2,coordinates= [x,y - 1,z - 1,t + 1]],
[complexRoots= ? - 2,coordinates= [x,y + 1,z - 1,t - 1]],
[complexRoots= ?,coordinates= [x,y + 1,z - 1,t - 1]],
[complexRoots= ? - 2,coordinates= [x - 1,y,z - 1,t - 1]],
[complexRoots= ?,coordinates= [x,y + 1,z - 1,t - 1]],

4 3 2
[complexRoots= ? + 5? + 16? + 30? + 57,
coordinates = 3 2 3 2
[151x + 15%A + 54%A + 104%A + 93, 151y - 10%A - 36%A - 19%A - 62,
3 2 3 2
151z - 5%A - 18%A - 85%A - 31, 151t - 5%A - 18%A - 85%A - 31]
]

, [complexRoots= ? + 2? - 8? + 48,
coordinates = 3 3
[8x - %A + 4%A - 8,2y + %A,8z + %A - 8%A + 8,8t - %A + 4%A - 8]
]

, [complexRoots= ? + 3? - 1,coordinates= [x - %A,y - %A,z - %A,t - %A]]

RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer))))
```

ts := lts.1
--R
--R  (22)
--R  2  3  3
--R \{t + t + 1, z - z - t + t,
--R
--R  3  2  2  3  6  3  3  2
--R (3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z
--R +
--R  6  3  9  6  3
--R (3t - 6t + 3)z + t - 3t + 5t - 3t
--R ,
--R x + y + z\}
--R Type: RegularChain(Integer,[x,y,z,t])
--E 22

--S 23 of 28
univariateSolve(ts)$pack
--R
--R  (23)
--R 4 3 2
--R [  
--R  [complexRoots= ? + 5? + 16? + 30? + 57,
--R  coordinates =  
--R  3  2  3  2  
--R [151x + 15%A + 54%A + 104%A + 93, 151y - 10%A - 36%A - 19%A - 62,
--R  3  2  3  2  
--R 151z - 5%A - 18%A - 85%A - 31, 151t - 5%A - 18%A - 85%A - 31]
--R ]
--R ,
--R 4 3 2
--R [complexRoots= ? - ? - 2? + 3,
--R 3 3 3
--R coordinates= \{x - %A + 2%A + 1, y + %A - %A - 1, z - %A, t + %A - %A - 1\}
--R ,
--R 4 3 2
--R [complexRoots= ? + 2? - 8? + 48,
--R coordinates =
--R 3 3 3
--R [8x - %A + 4%A - 8, 2y + %A, 8z + %A - 8%A + 8, 8t - %A + 4%A - 8]
--R ]
--R Type: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer)))
--E 23
realSolve(ts)$pack

\[
1 \quad 15 \quad 2 \quad 14 \quad 1 \quad 13 \quad 4 \quad 12 \quad 11 \quad 11 \quad 4 \quad 10
\]

\[
27 \quad 27 \quad 27 \quad 27 \quad 27
\]

\[
1 \quad 9 \quad 14 \quad 8 \quad 1 \quad 7 \quad 2 \quad 6 \quad 1 \quad 5 \quad 2 \quad 4 \quad 3
\]

\[
27 \quad 27 \quad 27 \quad 9 \quad 3 \quad 9
\]

\[
4 \quad 2
\]

\[
3
\]

\[
1 \quad 15 \quad 1 \quad 14 \quad 1 \quad 13 \quad 2 \quad 12 \quad 11 \quad 11 \quad 2 \quad 10
\]

\[
54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27
\]

\[
1 \quad 9 \quad 7 \quad 8 \quad 1 \quad 7 \quad 1 \quad 6 \quad 1 \quad 5 \quad 1 \quad 4 \quad 3
\]

\[
54 \quad 27 \quad 54 \quad 9 \quad 6 \quad 9
\]

\[
2 \quad 2 \quad 1 \quad 3
\]

\[
3 \quad 2 \quad 2
\]

\[
1 \quad 15 \quad 1 \quad 14 \quad 1 \quad 13 \quad 2 \quad 12 \quad 11 \quad 11 \quad 2 \quad 10
\]

\[
54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27
\]

\[
1 \quad 9 \quad 7 \quad 8 \quad 1 \quad 7 \quad 1 \quad 6 \quad 1 \quad 5 \quad 1 \quad 4 \quad 3
\]
```c
--R - -- %B32 - -- %B32 - -- %B32 - -- %B32 - -- %B32 - -- %B32 - -- %B32
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R - - %B32 + - %B32 + -
--R 3 2 2
--R ]
--R ,
--R
--R [%B33,
--R
--R 1 15 2 14 1 13 4 12 11 11 4 10
--R - -- %B33 + -- %B33 + -- %B33 + -- %B33 - -- %B33 - -- %B33
--R 27 27 27 27 27 27 27 27
--R +
--R 1 9 14 8 1 7 2 6 1 5 2 4 3
--R - -- %B33 + -- %B33 + -- %B33 + - %B33 + - %B33 + %B33
--R 27 27 27 9 3 9
--R +
--R 4 2
--R - %B33 - %B33 - 2
--R 3
--R ,
--R
--R 1 15 1 14 1 13 2 12 11 11 2 10
--R - -- %B33 - -- %B33 - -- %B33 + -- %B33 + -- %B33
--R 54 27 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R - -- %B33 - -- %B33 - -- %B33 - - %B33 - - %B33 - %B33
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R - %B33 + - %B33 + -
--R 3 2 2
--R ,
--R
--R 1 15 1 14 1 13 2 12 11 11 2 10
--R - -- %B33 - -- %B33 - -- %B33 + -- %B33 + -- %B33
--R 54 27 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R - -- %B33 - -- %B33 - -- %B33 - - %B33 - - %B33 - %B33
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R - %B33 + - %B33 + -
--R 3 2 2
--R ]
--R ,
```
| 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 | 1 | 4 | 3 |
| 54 | 27 | 54 | 9 | 6 | 9 |
| 2 | 2 | 1 | 3 |
| - | %B23 | + | %B23 | + | - |
| 3 | 2 | 2 |
| %B30, |
| 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| 54 | 27 | 54 | 27 | 54 | 27 |
| 2 | 10 | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 |
| - | %B23 | + | %B23 | + | %B23 | + | %B23 | + | %B23 | + | %B23 |
| 27 | 54 | 27 | 54 | 9 | 6 |
| 1 | 4 | 2 | 2 | 1 | 1 |
| - | %B23 | + | %B23 | - | %B23 | - | |
| 9 | 3 | 2 | 2 |
| %B31, |
| 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| 54 | 27 | 54 | 27 | 54 | 27 |
| 2 | 10 | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 |
| - | %B23 | + | %B23 | + | %B23 | + | %B23 | + | %B23 | + | %B23 |
| 27 | 54 | 27 | 54 | 9 | 6 |
| 1 | 4 | 2 | 2 | 1 | 1 |
| - | %B23 | + | %B23 | - | %B23 | - | |
CHAPTER 27. CHAPTER Z

--R 9 3 2 2
--R ]
--R ,
--R [\%B24,
--R
--R 1 15 1 14 1 13 2 12 11 11 2 10
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R +-- \%B24 +-- \%B24 +-- \%B24
--R 3 2 2
--R ,
--R \%B28,
--R
--R 1 15 1 14 1 13 2 12 11 11
--R --- \%B28 --- \%B24 --- \%B24 --- \%B24 --- \%B24 --- \%B24
--R 54 27 54 27 54
--R +
--R 2 10 1 9 7 8 1 7 1 6 1 5
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 27 54 27 54 9 6
--R +
--R 1 4 2 2 1 1
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 9 3 2 2
--R ]
--R ,
--R [\%B24,
--R
--R 1 15 1 14 1 13 2 12 11 11 2 10
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24 +-- \%B24
--R 3 2 2
--R ,
--R \%B29,
--R  1  15  1  14  1  13  2  12  11  11
--R  - %B29 +  %B24  +  %B24  +  %B24  -  %B24  -  %B24
--R     54    27    54    27    54
--R  +
--R  2  10  1  9  7  8  1  7  1  6  1  5
--R  -  %B24  +  %B24  +  %B24  +  %B24  +  %B24  +  %B24
--R     27    54    27    54    9    6
--R  +
--R  1  4  2  2  1  1
--R  -  %B24  +  %B24  -  %B24  -
--R     9    3    2    2
--R ]
--R ,
--R [ %B25,
--R ]
--R  1  15  1  14  1  13  2  12  11  11  2  10
--R  -  %B25  -  %B25  -  %B25  +  %B25  +  %B25  +  %B25
--R     54    27    54    27    54    27
--R +
--R  1  9  7  8  1  7  1  6  1  5  1  4  3
--R  -  %B25  -  %B25  -  %B25  -  %B25  -  %B25  -  %B25  -  %B25  -  %B25
--R     54    27    54    9    6    9
--R +
--R  2  2  1  3
--R  -  %B25  +  %B25  +
--R     3    2    2
--R ,
--R %B26,
--R ]
--R  1  15  1  14  1  13  2  12  11  11
--R  -  %B26  +  %B25  +  %B25  +  %B25  -  %B25  -  %B25
--R     54    27    54    27    54
--R +
--R  2  10  1  9  7  8  1  7  1  6  1  5
--R  -  %B25  +  %B25  +  %B25  +  %B25  +  %B25  +  %B25
--R     27    54    27    54    9    6
--R +
--R  1  4  2  2  1  1
--R  -  %B25  +  %B25  -  %B25  -
--R     9    3    2    2
--R ]
--R ,
--R [ %B25,
--R ]
```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(27)</td>
<td>[%B40, - %B40 + -,- - %B40 + -]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type: List(List(RealClosure(Fraction(Integer))))
```
--- S 28 of 28
[approximate(r,1/10**21)::Float for r in lpr2.1]
--R
--R
--R (28)
--R [0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291,
--R 0.3221853546 2608559291]
--R
--E 28
)spool
)lisp (bye)

---

--- ZeroDimensionalSolvePackage.help ---

====================================================================
ZeroDimensionalSolvePackage examples
====================================================================

A package for computing symbolically the complex and real roots of zero-dimensional algebraic systems over the integer or rational numbers. Complex roots are given by means of univariate representations of irreducible regular chains. Real roots are given by means of tuples of coordinates lying in the RealClosure of the coefficient ring.

This constructor takes three arguments. The first one R is the coefficient ring. The second one ls is the list of variables involved in the systems to solve. The third one must be concat(ls,s) where s is an additional symbol used for the univariate representations.

WARNING. The third argument is not checked.

All operations are based on triangular decompositions. The default is to compute these decompositions directly from the input system by using the RegularChain domain constructor.

The lexTriangular algorithm can also be used for computing these decompositions (see LexTriangularPackage package constructor).

For that purpose, the operations univariateSolve, realSolve and positiveSolve admit an optional argument.

The ZeroDimensionalSolvePackage package constructor provides operations for computing symbolically the complex or real roots of zero-dimensional algebraic systems.

The package provides no multiplicity information (i.e. some returned
roots may be double or higher) but only distinct roots are returned.

Complex roots are given by means of univariate representations of irreducible regular chains. These representations are computed by the \texttt{univariateSolve} operation (by calling the \texttt{InternalRationalUnivariateRepresentationPackage} package constructor which does the job).

Real roots are given by means of tuples of coordinates lying in the \texttt{RealClosure} of the coefficient ring. They are computed by the \texttt{realSolve} and \texttt{positiveSolve} operations. The former computes all the solutions of the input system with real coordinates whereas the later concentrate on the solutions with (strictly) positive coordinates. In both cases, the computations are performed by the \texttt{RealClosure} constructor.

Both computations of complex roots and real roots rely on triangular decompositions. These decompositions can be computed in two different ways. First, by applying the \texttt{zeroSetSplit} operation from the \texttt{REGSET} domain constructor. In that case, no Groebner bases are computed. This strategy is used by default. Secondly, by applying the \texttt{zeroSetSplit} from \texttt{LEXTRIPK}. To use this later strategy with the operations \texttt{univariateSolve}, \texttt{realSolve} and \texttt{positiveSolve} one just needs to use an extra boolean argument.

Note that the way of understanding triangular decompositions is detailed in the example of the \texttt{RegularTriangularSet} constructor.

The \texttt{ZeroDimensionalSolvePackage} constructor takes three arguments. The first one \(R\) is the coefficient ring; it must belong to the categories \texttt{OrderedRing}, \texttt{EuclideanDomain}, \texttt{CharacteristicZero} and \texttt{RealConstant}. This means essentially that \(R\) is \texttt{Integer} or \texttt{Fraction(Integer)}. The second argument \(ls\) is the list of variables involved in the systems to solve. The third one MUST BE \texttt{concat(ls,s)} where \(s\) is an additional symbol used for the univariate representations. The abbreviation for \texttt{ZeroDimensionalSolvePackage} is \texttt{ZDSOLVE}.

We illustrate now how to use the constructor \texttt{ZDSOLVE} by two examples: the Arnborg and Lazard system and the L-3 system (Aubry and Moreno Maza). Note that the use of this package is also demonstrated in the example of the \texttt{LexTriangularPackage} constructor.

Define the coefficient ring.

\begin{verbatim}
R := Integer
Integer
Type: Domain
\end{verbatim}

Define the lists of variables:

\begin{verbatim}
ls : List Symbol := [x,y,z,t]
\end{verbatim}
and:

\[
\text{ls2 : List Symbol := [x,y,z,t,new()$Symbol]}
\]

\[
[x,y,z,t,%A]
\]

Type: List Symbol

Call the package:

\[
\text{pack := ZDSOLVE(R,ls,ls2)}
\]

ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])

Type: Domain

Define a polynomial system (Arnborg-Lazard)

\[
p1 := x**2*y*y*z + x*y**2*z + x*y*z**2 + x*y*z + x + y*z + x**2 + y + z + x y z + (x y + (x + x + 1)y + x)z + x y
\]

Type: Polynomial Integer

\[
p2 := x**2*y**2*z + x**2*y**2*z**2 + x**2*y**2*z + x*y*z + y*z + x + z
\]

Type: Polynomial Integer

\[
p3 := x**2*y**2*z**2 + x**2*y**2*z + x*y**2*z + x*y*z + x*z + z + 1
\]

Type: Polynomial Integer

\[
lp := [p1, p2, p3]
\]

\[
[x y z + (x y + (x + x + 1)y + x)z + x y,
\]

Type: List Polynomial Integer

Note that these polynomials do not involve the variable \( t \); we will use it in the second example.

First compute a decomposition into regular chains (i.e. regular triangular sets).

\[
\text{triangSolve(lp)$pack}
\]

[ ]
We can see easily from this decomposition (consisting of a single regular chain) that the input system has 20 complex roots.

Then we compute a univariate representation of this regular chain.

```plaintext
type: List RegularChain(Integer,[x,y,z,t])
```

We can see easily from this decomposition (consisting of a single regular chain) that the input system has 20 complex roots.

Then we compute a univariate representation of this regular chain.
We see that the zeros of our regular chain are split into three components. This is due to the use of univariate polynomial factorization.

Each of these components consist of two parts. The first one is an irreducible univariate polynomial \( p(?) \) which defines a simple algebraic extension of the field of fractions of \( R \). The second one consists of multivariate polynomials \( \text{pol1}(x,?A), \text{pol2}(y,?A) \) and \( \text{pol3}(z,?A) \). Each of these polynomials involve two variables: one is an indeterminate \( x, y \) or \( z \) of the input system \( lp \) and the other is \( ?A \) which represents any root of \( p(?) \). Recall that this \( ?A \) is the last element of the third parameter of \( \text{ZDSOLVE} \). Thus any complex root \( ? \) of \( p(?) \) leads to a solution of the input system \( lp \) by replacing \( ?A \) by this \( ? \) in \( \text{pol1}(x,?A), \text{pol2}(y,?A) \) and \( \text{pol3}(z,?A) \). Note that the polynomials \( \text{pol1}(x,?A), \text{pol2}(y,?A) \) and \( \text{pol3}(z,?A) \) have degree one w.r.t. \( x, y \) or \( z \) respectively. This is always the case for all
univariate representations. Hence the operation univariateSolve replaces a system of multivariate polynomials by a list of univariate polynomials, what justifies its name. Another example of univariate representations illustrates the LexTriangularPackage package constructor.

We now compute the solutions with real coordinates:

```plaintext
lr := realSolve(lp)$pack
[
  [%B1,
   1184459 19 2335702 18 5460230 17 7900378 16
   ------- %B1 - ------- %B1 - ------- %B1 + ------- %B1
   1645371 548457 182819 1645371
  +
   43953929 15 13420192 14 553986 13 193381378 12
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   548457 182819 3731 1645371
  +
   35978916 11 35860781 10 271667666 9 118784873 8
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   182819 1645371 1645371 548457
  +
   337505020 7 1389370 6 688291 5 193381378 4
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   1645371 343 1645371
  +
   140671876 3 32325724 2 8270 9741532
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   1645371 548457 1645371
  ,
   91729 19 487915 18 4114333 17 1276987 16
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   705159 705159 705159 235053
  +
   13243117 15 16292173 14 26536060 13 722714 12
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   705159 705159 705159 18081
  +
   5382578 11 15449995 10 142797770 9 6603890 8
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   100737 235053 235053 100737
  +
   409930 7 37340389 6 34893715 5 26686318 4
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   6027 705159 705159 705159
  +
   801511 3 17206178 2 4406102 377534
   ------- %B1 + ------- %B1 + ------- %B1 + ------- %B1
   26117 705159 705159 705159
]```
[\%B2,  
1184459  19  2335702  18  5460230  17  79900378  16  
1645371  548457  182819  1645371  
+  
43953929  15  13420192  14  553986  13  193381378  12  
548457  182819  3731  1645371  
+  
35978916  11  368660781  10  271667666  9  118784873  8  
182819  1645371  1645371  548457  
+  
337505020  7  1389370  6  688291  5  3378002  4  
1645371  11193  4459  42189  
+  
140671876  3  32325724  2  8270  9741532  
1645371  548457  343  1645371  
]  
,  
[\%B3,  
1184459  19  2335702  18  5460230  17  79900378  16  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
1645371  548457  182819  1645371  
+  
43953929  15  13420192  14  553986  13  193381378  12  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
43953929  15  13420192  14  553986  13  193381378  12  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
1645371  548457  182819  1645371  
+  
35978916  11  368660781  10  271667666  9  118784873  8  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
182819  1645371  1645371  548457  
+  
337505020  7  1389370  6  688291  5  3378002  4  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
1645371  11193  4459  42189  
+  
140671876  3  32325724  2  8270  9741532  
-------- \%B3  +  +  +  \%B3  +  +  \%B3  +  +  \%B3  
1645371  548457  343  1645371  
]
```
<table>
<thead>
<tr>
<th></th>
<th>548457</th>
<th>182819</th>
<th>3731</th>
<th>1645371</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>35978916</td>
<td>11</td>
<td>358660781</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>182819</td>
<td>1645371</td>
<td>1645371</td>
<td>548457</td>
</tr>
<tr>
<td>+</td>
<td>337505020</td>
<td>7</td>
<td>1389370</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>1645371</td>
<td>11193</td>
<td>4459</td>
<td>42189</td>
</tr>
<tr>
<td>+</td>
<td>140671876</td>
<td>3</td>
<td>32325724</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>1645371</td>
<td>548457</td>
<td>343</td>
<td>1645371</td>
</tr>
<tr>
<td>,</td>
<td>91729</td>
<td>19</td>
<td>487915</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>235053</td>
</tr>
<tr>
<td>+</td>
<td>13243117</td>
<td>15</td>
<td>16292173</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>18081</td>
</tr>
<tr>
<td>+</td>
<td>5382578</td>
<td>11</td>
<td>15449995</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>100737</td>
<td>235053</td>
<td>235053</td>
<td>100737</td>
</tr>
<tr>
<td>+</td>
<td>409930</td>
<td>7</td>
<td>37340389</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>6027</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
</tr>
<tr>
<td>+</td>
<td>801511</td>
<td>3</td>
<td>17206178</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>---%B3</td>
<td>+------</td>
<td>---%B3</td>
<td>+------</td>
</tr>
<tr>
<td></td>
<td>26117</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
</tr>
<tr>
<td>]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
, [%B4,
1184459 | 19     | 2335702 | 18     | 5460230 | 17    | 79900378 | 16    |
|       | ---%B4 | +------ | ---%B4  | +------ | ---%B4  | +------ | ---%B4  |
|       | 1645371 | 548457 | 182819 | 1645371 |
| +     | 43953929 | 15     | 13420192 | 14    | 553986  | 13    | 193381378 | 12    |
|       | ---%B4  | +------ | ---%B4  | +------ | ---%B4  | +------ | ---%B4  |
|       | 548457 | 182819 | 3731   | 1645371 |
| +     | 35978916 | 11     | 358660781 | 10    | 271667666 | 9     | 118784873 | 8     |
|       | ---%B4  | +------ | ---%B4  | +------ | ---%B4  | +------ | ---%B4  |
|       | 182819 | 1645371 | 1645371 | 548457  |
| +     | 337505020 | 7      | 1389370  | 6      | 688291  | 5     | 3378002  | 4     |
|       | ---%B4  | +------ | ---%B4  | +------ | ---%B4  | +------ | ---%B4  |
```

\[
\begin{align*}
&1645371 & 11193 & 4459 & 42189 \\
+ & 140671876 & 3 & 32325724 & 2 & 8270 & 9741532 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 1645371 & 548457 & 343 & 1645371 \\
& , & 91729 & 19 & 487915 & 18 & 4114333 & 17 & 1276987 & 16 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 705159 & 705159 & 705159 & 235053 \\
+ & 13243117 & 15 & 16292173 & 14 & 26536060 & 13 & 722714 & 12 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 705159 & 705159 & 705159 & 18081 \\
+ & 5382578 & 11 & 15449995 & 10 & 14279770 & 9 & 6603890 & 8 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 100737 & 235053 & 235053 & 100737 \\
+ & 409930 & 7 & 37340389 & 6 & 34893715 & 5 & 26686318 & 4 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 6027 & 705159 & 705159 & 705159 \\
+ & 801511 & 3 & 17206178 & 2 & 4406102 & 377534 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 26117 & 705159 & 705159 & 705159 \\
\right]
\]
\[
\begin{align*}
& \% \text{B5}, \\
& 1184459 & 19 & 2335702 & 18 & 5460230 & 17 & 79900378 & 16 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 1645371 & 548457 & 182819 & 1645371 \\
+ & 43953929 & 15 & 13420192 & 14 & 553986 & 13 & 193381378 & 12 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 548457 & 182819 & 3731 & 1645371 \\
+ & 35978916 & 11 & 358660781 & 10 & 271667666 & 9 & 118784873 & 8 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 182819 & 1645371 & 1645371 & 548457 \\
+ & 337505020 & 7 & 1389370 & 6 & 688291 & 5 & 3378002 & 4 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 1645371 & 11193 & 4459 & 42189 \\
+ & 140671876 & 3 & 32325724 & 2 & 8270 & 9741532 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 1645371 & 548457 & 343 & 1645371 \\
\right]
\]
\[
\begin{align*}
& 91729 & 19 & 487915 & 18 & 4114333 & 17 & 1276987 & 16 \\
& - & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} \\
& 1645371 & 11193 & 4459 & 42189 \\
\end{align*}
\]
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>235053</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13243117</td>
<td>15</td>
<td>16292173</td>
<td>14</td>
<td>26536060</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5382578</td>
<td>11</td>
<td>15449995</td>
<td>10</td>
<td>14279770</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>409930</td>
<td>7</td>
<td>37340389</td>
<td>6</td>
<td>34893715</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>801511</td>
<td>3</td>
<td>17206178</td>
<td>2</td>
<td>4406102</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1184459</td>
<td>19</td>
<td>2335702</td>
<td>18</td>
<td>5460230</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43953929</td>
<td>15</td>
<td>13420192</td>
<td>14</td>
<td>553986</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35978916</td>
<td>11</td>
<td>358660781</td>
<td>10</td>
<td>271667666</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140671876</td>
<td>3</td>
<td>32325724</td>
<td>2</td>
<td>8270</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13243117</td>
<td>15</td>
<td>16292173</td>
<td>14</td>
<td>26536060</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5382578</td>
<td>11</td>
<td>15449995</td>
<td>10</td>
<td>14279770</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[
\begin{array}{cccc}
100737 & 235053 & 235053 & 100737 \\
+ & 409930 & 7 & 37340389 & 6 & 34893715 & 5 & 26686318 & 4 \\
\quad - & - - - - - & \%B6 & - & - - - - & \%B6 & - & - - - - & \%B6 & - & - - - - & \%B6 \\
\quad 6027 & 705159 & 705159 & 705159 \\
+ & 801511 & 3 & 17206178 & 2 & 4406102 & 377534 \\
\quad - & - - - - & \%B6 & - & - - - - & \%B6 & - & - - - - & \%B6 & + & - & - \\
\quad 26117 & 705159 & 705159 & 705159 \\
\end{array}
\]

\[
\begin{array}{cccc}
\end{array}
\]

\[
\begin{array}{cccc}
1184459 & 19 & 2335702 & 18 & 5460230 & 17 & 79900378 & 16 \\
\quad - & - - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 & + & - - - - & \%B7 \\
\quad 1645371 & 548457 & 182819 & 1645371 \\
+ & 43953929 & 15 & 13420192 & 14 & 553986 & 13 & 193381378 & 12 \\
\quad - & - - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 \\
\quad 548457 & 182819 & 3731 & 1645371 \\
+ & 35978916 & 11 & 358660781 & 10 & 271667666 & 9 & 118784873 & 8 \\
\quad - & - - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 \\
\quad 1645371 & 1645371 & 1645371 & 548457 \\
+ & 337505020 & 7 & 1389370 & 6 & 688291 & 5 & 3378002 & 4 \\
\quad - & - - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 \\
\quad 1645371 & 11193 & 4459 & 42189 \\
+ & 140671876 & 3 & 32325724 & 2 & 8270 & 9741532 \\
\quad - & - - - - - & \%B7 & + & - - - - & \%B7 & - & - - & \%B7 & - & - & - & - \\
\quad 1645371 & 548457 & 343 & 1645371 \\
, \\
91729 & 19 & 487915 & 18 & 4114333 & 17 & 1276987 & 16 \\
\quad - & - - - - - & \%B7 & + & - - - - & \%B7 & + & - - - - & \%B7 & - & - - - - & \%B7 \\
\quad 705159 & 705159 & 705159 & 235053 \\
+ & 13243117 & 15 & 16292173 & 14 & 26536060 & 13 & 722714 & 12 \\
\quad - & - - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 \\
\quad 705159 & 705159 & 705159 & 18081 \\
+ & 5382678 & 11 & 15449995 & 10 & 14279770 & 9 & 6603890 & 8 \\
\quad - & - - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 \\
\quad 100737 & 235053 & 235053 & 100737 \\
+ & 409930 & 7 & 37340389 & 6 & 34893715 & 5 & 26686318 & 4 \\
\quad - & - - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 \\
\quad 6027 & 705159 & 705159 & 705159 \\
+ & 801511 & 3 & 17206178 & 2 & 4406102 & 377534 \\
\quad - & - - - - - & \%B7 & - & - - - - & \%B7 & - & - - - - & \%B7 & + & - \\
\end{array}
\]
The number of real solutions for the input system is:

```
# lr
```
Each of these real solutions is given by a list of elements in RealClosure(R). In these 8 lists, the first element is a value of z, the second of y and the last of x. This is logical since by setting the list of variables of the package to [x,y,z,t] we mean that the elimination ordering on the variables is t < z < y < x. Note that each system treated by the ZDSOLVE package constructor needs only to be zero-dimensional w.r.t. the variables involved in the system itself and not necessarily w.r.t. all the variables used to define the package.

We can approximate these real numbers as follows. This computation takes between 30 sec. and 5 min, depending on your machine.

```plaintext
[ [approximate(r,1/1000000) for r in point] for point in lr]
[
  10048059
  [- --------,
   2097152
   4503057283025245488516511806985826635083100693757320465280554706965644,
   94977509916867201889438090408354817931718593852797624551518983570973,
   04877442429148870829840324189200301436123314860200821443733790755311,
   24363291986499421704228949512900161119498087957023663865443069392027,
   14897986829671232356043491523434068924275280417338574817381189277066,
   143312396681216,
   2106260768234750738947988680486016596249607148690685538763683715020639,
   68085864965079005688950564689330447097099937802187329095325898785247,
   2490207150498366048207515661873872451468533306001120296435166381351,
   54325598220025030528398108687110614842307026901211297929876896256851,
   83047905476005638076266490561846205306047816191782011588703789138988,
   1895]
]
```

437701
[-        ,
2097152

1683106908638349588322172332654225913562986313181951031452750161441497_
47345532815072136468835557964678160350777199075077835213366484533654_
91383623741304759

168310686809521338900170998270591363863077668731226111167785188004907_
42522629868032588787109626141402985973669842648879989083770687899998454_
2338164900809328

4961551098350101864226810134221087359587148010037606397097968096646912_
8267084728344311723917219104249213450966312411133

496154987275773831509192078210209029852897118611097126236384040829376_
59261914313170254867464792718363492160482442215424

222801
[--------,
2097152

- 8994884880402428265107595121970691427136045692541978275573001865213_
75992158813716696126349101655220195142994932299137183241705867672_
383477

/ 116789999665026372177785100691888582708969960229934769690835752457_
0777794164352094737376786650776940588942764587718542434255625992456_
372224,

- 2389704888133156878320801544373808395612771509208491019847452991885_
50954651952564783901661359399693866640036283570552321155037871291_
458705265

/ 535548727364509632609040328668993190598225444685411433221593833681_
1929575628336714688654290340746993656285925599117602120446183443145_
479421952

765693
[--------,
2097152
We can also concentrate on the solutions with real (strictly) positive coordinates:

```plaintext
Type: List List RealClosure Fraction Integer
```

We can also concentrate on the solutions with real (strictly) positive coordinates:

```plaintext
lpr := positiveSolve(lp)$pack
[]
```

Type: List List RealClosure Fraction Integer
Thus we have checked that the input system has no solution with strictly positive coordinates.

Let us define another polynomial system (L-3).

\[
\begin{align*}
\text{lts} & := \text{triangSolve}(\text{lfs})$pack \\

f0 & := x^3 + y + z + t - 1 \\
    & \text{Type: Polynomial Integer} \\
\end{align*}
\]

\[
\begin{align*}
f1 & := x + y^3 + z + t - 1 \\
    & \text{Type: Polynomial Integer} \\
\end{align*}
\]

\[
\begin{align*}
f2 & := x + y + z^3 + t - 1 \\
    & \text{Type: Polynomial Integer} \\
\end{align*}
\]

\[
\begin{align*}
f3 & := x + y + z + t^3 - 1 \\
    & \text{Type: Polynomial Integer} \\
\end{align*}
\]

\[
\begin{align*}
lfs & := \left[ f0, f1, f2, f3 \right] \\
    & \text{Type: List Polynomial Integer} \\
\end{align*}
\]

First compute a decomposition into regular chains (i.e. regular triangular sets).

\[
\begin{align*}
lts & := \text{triangSolve}(lfs)$pack \\

\end{align*}
\]
\[\begin{align*}
&+ 10 9 8 7 \\
&162959040t - 159859440t - 156802608t + 117168768t \\
&+ 6 5 4 3 \\
&126282384t - 129351600t + 306646992t + 475302816t \\
&+ 2 \\
&- 1006837776t - 237269088t + 480716208 \\
&\ast \\
&+ z \\
&+ 54 51 48 46 45 43 42 \\
&48t - 912t + 8232t - 72t - 46848t + 1152t + 186324t \\
&+ 40 39 38 37 36 35 \\
&126282384t - 129351600t + 306646992t + 475302816t \\
&+ 34 33 32 31 30 \\
&1427803t - 1843242t - 301815t - 1440726t + 1912012t \\
&+ 29 28 27 26 25 \\
&1442826t + 4696262t - 922481t - 4816188t - 10583524t \\
&+ 24 23 22 21 20 \\
&- 208751t + 11472138t + 16762859t - 857663t - 19328175t \\
&+ 19 18 17 16 15 \\
&- 18270421t + 4914903t + 22483044t + 12926517t - 8605511t \\
&+ 14 13 12 11 10 \\
&- 17455518t - 5014597t + 8108814t + 8465535t + 190542t \\
&+ 9 8 7 6 5 4 \\
&- 4305624t - 2226123t + 661905t + 1169775t + 226260t - 299952t \\
&+ 3 \\
&- 141183t + 27216t \\
&\bigl(3z + 3t - 3\bigr)y + \bigl(3z + \bigl(6t - 6\bigr)z + 3t - 6t + 3\bigr)y + \bigl(3t - 3\bigr)z \\
&+ 6 3 9 6 3 \\
&\bigl(3t - 6t + 3\bigr)z + t - 3t + 5t - 3t \\
&+ 3 \\
&x + y + z + t - 1 \\
&\bigl\{t, z - 1, y - 1, x + y\bigr\}, \bigl\{t - 1, z, y - 1, x + y\bigr\}, \bigl\{t - 1, z - 1, z y + 1, x\bigr\},
\end{align*}\]
\[ t - 6t + 9t - 4t + 15t - 54t + 27, \]
\[ 4907232t + 40899384t - 115013088t - 1730448t - 168139584t \]
\[ + 738024480t - 195372208t + 315849456t - 256729232t \]
\[ + 937147968t + 1026357696t + 4780488240t - 2893767696t \]
\[ + 1617160352t - 342765128t + 5001100848t + 8720098416t \]
\[ + 5617160352t - 342765128t + 5001100848t + 8720098416t \]
\[ + 2331732960t - 499046544t - 16243306272t - 9748123200t \]
\[ + 3927244320t + 25257280896t + 10348032096t - 17128072128t \]
\[ + 14755488768t + 544086720t + 10848188736t + 1423614528t \]
\[ + 2884297248 \]
\[ * z \]
\[ - 48t + 1152t - 13560t + 103656t - 7560t - 572820t \]
\[ + 54 \]
\[ 71316t + 2414556t + 2736t - 402876t - 7985131t - 49248t \]
\[ + 48 \]
\[ 1431133t + 20977409t + 521487t - 697635t - 43763654t \]
\[ + 43 \]
\[ 3756573t - 2093410t + 71546495t + 19699032t + 35025028t \]
\[ + 38 \]
\[ 89623786t - 77798760t - 138654191t + 87596128t + 235642497t \]
\[ + 33 \]
\[ 349607642t - 93299834t - 551563167t - 630995176t + 186818962t \]
\[ + 28 \]
\[ 995427468t + 828416204t - 393919231t - 1076617485t \]
\[ + 24 \]
\[ - 1609479791t + 595738126t + 1198787136t + 4342832069t \]
\[
\begin{align*}
&+20 19 18 17 \\
&\quad - 2075938757t - 4390835799t - 4822843033t + 6932747678t \\
&+16 15 14 13 \\
&\quad 6172196808t + 1141517740t - 4981677585t - 9819815280t \\
&+12 11 10 9 \\
&\quad - 7404299976t - 157295760t + 29124027630t + 14856038208t \\
&+8 7 6 5 \\
&\quad - 16184101410t - 26935440354t - 3574164258t + 10271338974t \\
&+4 3 2 \\
&\quad 11191425264t + 6869861262t - 9780477840t - 3586674168t + 2884297248 \\
&3 3 2 6 3 9 6 3 \\
&\quad (3z + (6t - 6)z + (6t - 12t + 3)z + 2t - 6t + t + 3t)y \\
&+3 3 6 3 2 9 6 3 12 9 \\
&\quad (3t - 3)z + (6t - 12t + 6)z + (4t - 12t + 11t - 3)z + t - 4t \\
&+6 3 \\
&\quad 5t - 2t \\
&3 x + y + z + t - 1} \\
&\{t - 1, z - 1, y, x + z}, \\
&8 7 6 5 4 3 2 \\
&\{t + t + t - 2t - 2t - 2t + 19t + 19t - 8, \\
&\quad 7 6 5 4 3 \\
&\quad 2395770t + 3934440t - 3902067t - 10084164t - 1010448t \\
&+2 \\
&\quad 32386932t + 22413225t - 10432368 \\
* \\
&\quad z \\
+ \\
&\quad 7 6 5 4 3 \\
&\quad - 463519t + 3586833t + 9494955t - 8539305t - 33283098t \\
+ \\
&\quad 2 \\
&\quad 35479377t + 46263256t - 17419896 \\
\quad - 4 3 3 6 3 2 3 \\
&\quad 3z + (9t - 9)z + (12t - 24t + 9)z + (152t + 219t - 67)z \\
+ \\
&\quad 6 4 3 \\
\end{align*}
\]
Then we compute a univariate representation.

```
univariateSolve(lf)$pack
[[complexRoots= ?,coordinates= [x - 1,y - 1,z + 1,t - %A]],
 [complexRoots= ?,coordinates= [x,y - 1,z,t - %A]],
 [complexRoots= ? - 1,coordinates= [x,y,z,t - %A]],
 [complexRoots= ?,coordinates= [x - 1,y,z,t - %A]],
 [complexRoots= ?,coordinates= [x,y,z - 1,t - %A]],
 [complexRoots= ?,coordinates= [x - 1,y,z - 1,t]],
 [complexRoots= ?,coordinates= [x - 1,y,z,t]],

  6  3  2
 [complexRoots= ? - 2? + 3? - 3,
   3  3
 coordinates= [2x + %A + %A - 1,2y + %A + %A - 1,z - %A,t - %A]]
 ,

  5  3  2
 [complexRoots= ? + 3? - 2? + 3? - 3,
   3
 coordinates= [x - %A,y - %A,z + %A + 2%A - 1,t - %A]]
 ,

  4  3  2
 [complexRoots= ? - ? - 2? + 3,
   3  3
 coordinates= [x + %A - %A - 1,y + %A - %A - 1,z - %A + 2%A + 1,t - %A]]
 ,
```
[complexRoots= ? + 1, coordinates= [x - 1, y - 1, z, t - %A]],

6 3 2
[complexRoots= ? + 2? + 3? - 3,
3 coordinates= [2x - %A - %A - 1, y + %A, 2z - %A - %A - 1, t + %A]]

6 4 3 2
coordinates =
5 4 3 2
[12609x + 23%A + 49%A - 46%A + 362%A - 5015%A - 8239,
5 4 3 2
25218y + 23%A + 49%A - 46%A + 362%A + 7594%A - 8239,
5 4 3 2
25218z + 23%A + 49%A - 46%A + 362%A + 7594%A - 8239,
5 4 3 2
12609t + 23%A + 49%A - 46%A + 362%A - 5015%A - 8239]}

5 3 2
3 coordinates= [8x + %A + 8%A - 8, 2y - %A, 2z - %A, 2t - %A]]

5 4 3 2
coordinates =
3 3 3
[2x - %A + 2%A - 1, 2y + %A - 4%A + 1, 2z - %A + 2%A - 1,
3 3
2t - %A + 2%A - 1]]

4 3 2
[complexRoots= ? - 3? + 4? - 6? + 13,
coordinates =
3 2 3 2
[9x - 2%A + 4%A - %A + 2, 9y + %A - 2%A + 5%A - 1,
3 2 3 2
9z + %A - 2%A + 5%A - 1, 9t + %A - 2%A - 4%A - 1]]
4 2
[complexRoots= ? - 11? + 37,
coordinates =
2 2 2 2 2
[3x - %A + 7,6y + %A + 3%A - 7,3z - %A + 7,6t + %A - 3%A - 7]
]

, [complexRoots= ? + 1,coordinates= [x - 1,y,z - 1,t + 1]],
[complexRoots= ? + 2,coordinates= [x,y - 1,z - 1,t + 1]],
[complexRoots= ? - 2,coordinates= [x,y - 1,z + 1,t - 1]],
[complexRoots= ?,coordinates= [x,y + 1,z - 1,t - 1]],
[complexRoots= ?,coordinates= [x + 1,y,z - 1,t - 1]],

4 3 2
[complexRoots= ? + 5? + 16? + 30? + 57,
coordinates =
3 2 3 2 3 2
[151x + 15%A + 54%A + 104%A + 93, 151y - 10%A - 36%A - 19%A - 62,
3 2 3 2
151z - 5%A - 18%A - 85%A - 31, 151t - 5%A - 18%A - 85%A - 31]
]

, [complexRoots= ? - ? - 2? + 3,
coordinates= [x - %A + 2%A + 1,y + %A - %A - 1,z - %A,t + %A - %A - 1]]

, 4 3 2
[complexRoots= ? + 2? - 8? + 48,
coordinates =
3 3 3
[8x - %A + 4%A - 8,2y + %A,8z + %A - 8%A + 8,8t - %A + 4%A - 8]
]

, [complexRoots= ? + ? - 2? - 4? + 5? + 8,
coordinates= [3x + %A - 1,3y + %A - 1,3z + %A - 1,t - %A]]

, 3
[complexRoots= ? + 3? - 1,coordinates= [x - %A,y - %A,z - %A,t - %A]]

Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
Note that this computation is made from the input system \( l \).

However it is possible to reuse a pre-computed regular chain as follows:

\[
ts := lts.1
t = t + t + 1, z = z - t + t,
\]

\[
\begin{align*}
(3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z \\
+ (6t - 6t + 3)z + t - 3t + 5t - 3t,
\end{align*}
\]

\( x + y + z \)

Type: RegularChain(Integer,[x,y,z,t])

univariateSolve(ts)$pack

\[
\begin{align*}
[& \text{complexRoots: } ? + 5? + 16? + 30? + 57, \\
& \text{coordinates: } \{151x + 15%A + 54%A + 104%A + 93, 151y - 10%A - 36%A - 19%A - 62, \\
& 151z - 5%A - 18%A - 85%A - 31, 151t - 5%A - 18%A - 85%A - 31\} ]
\end{align*}
\]

\[
[& \text{complexRoots: } ? - ? - 2? + 3, \\
& \text{coordinates: } \{x - %A + 2%A + 1,y + %A - %A - 1,z - %A,t + %A - %A - 1\} ]
\]

\[
[& \text{complexRoots: } ? + 2? - 8? + 48, \\
& \text{coordinates: } \{8x - %A + 4%A - 8,2y + %A,8z + %A - 8%A + 8,8t - %A + 4%A - 8\} ]
\]

Type: List Record(complexRoots: SparseUnivariatePolynomial Integer, coordinates: List Polynomial Integer)

realSolve(ts)$pack

\[
[&]
\]

Type: List List RealClosure Fraction Integer

We compute now the full set of points with real coordinates:
\[ lr2 := \text{realSolve}(l)$pack \]
\[
[[0, -1, 1, 1], [0, 0, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1], [1, 0, \%B37, - \%B37],
[1, 0, \%B38, - \%B38], [0, 1, \%B35, - \%B35], [0, 1, \%B35, - \%B36], [-1, 0, 1, 1],
[\%B32,
1 15 2 14 1 13 4 12 11 11 4 10
-- \%B32 + -- \%B32 + -- \%B32 -- \%B32 -- \%B32 -- \%B32
27 27 27 27 27 27
+ 1 9 14 8 1 7 2 6 1 5 2 4 3
-- \%B32 + -- \%B32 + -- \%B32 + - \%B32 + - \%B32 + - \%B32 + \%B32
27 27 27 9 3 9
+ 4 2
- \%B32 - \%B32 - 2
3
, 1 15 1 14 1 13 2 12 11 11 2 10
-- \%B32 -- \%B32 -- \%B32 + -- \%B32 + -- \%B32 + -- \%B32
54 27 54 27 54 27
+ 1 9 7 8 1 7 1 6 1 5 1 4 3
-- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32
54 27 54 9 6 9
+ 2 2 1 3
- - \%B32 + - \%B32 + -
3 2 2
, 1 15 1 14 1 13 2 12 11 11 2 10
-- \%B32 -- \%B32 -- \%B32 + -- \%B32 + -- \%B32 + -- \%B32
54 27 54 27 54 27
+ 1 9 7 8 1 7 1 6 1 5 1 4 3
-- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32 -- \%B32
54 27 54 9 6 9
+ 2 2 1 3
- - \%B32 + - \%B32 + -
3 2 2
]

[\%B33,
1 15 2 14 1 13 4 12 11 11 4 10
-- \%B33 + -- \%B33 + -- \%B33 -- \%B33 -- \%B33 -- \%B33
27 27 27 27 27 27
+ 1 9 14 8 1 7 2 6 1 5 2 4 3
-- \%B33 + -- \%B33 + -- \%B33 + - \%B33 + - \%B33 + - \%B33 + \%B33
27 27 27 9 3 9
]
+ 1 9 7 8 1 7 1 6 1 5 1 4 3
- -- %B23 - -- %B23 - -- %B23 - -- %B23 - -- %B23 - -- %B23
54 27 54 9 6 9
+ 2 2 1 3
- - %B23 + - %B23 + - %B23 + - %B23
3 2 2

] %B31,
1 15 1 14 1 13 2 12 11 11
- %B31 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23
54 27 54 27 54
+ 2 10 1 9 7 8 1 7 1 6 1 5
- -- %B23 + -- %B23 + -- %B23 + -- %B23 + - %B23 + - %B23
27 54 27 54 9 6
+ 1 4 2 2 1 1
- %B23 + - %B23 + - %B23
9 3 2 2

] %B24,
1 15 1 14 1 13 2 12 11 11 2 10
- -- %B24 - -- %B24 - -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24
54 27 54 27 54 27
+ 1 9 7 8 1 7 1 6 1 5 1 4 3
- -- %B24 - -- %B24 - -- %B24 - -- %B24 - -- %B24 - -- %B24 - -- %B24
54 27 54 9 6 9
+ 2 2 1 3
- - %B24 + - %B24 + - %B24 + - %B24 + - %B24 + - %B24
3 2 2

] %B28,
1 15 1 14 1 13 2 12 11 11
- %B28 + -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24
54 27 54 27 54
+ 2 10 1 9 7 8 1 7 1 6 1 5
- -- %B24 + -- %B24 + -- %B24 + -- %B24 + - %B24 + - %B24
27 54 27 54 9 6
+ 1 4 2 2 1 1
- %B24 + - %B24 + - %B24
9 3 2 2

]}
| [%B24 | 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- |
| + | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 | 1 | 4 | 3 |
| -- [%B24 | -- [%B24 | -- [%B24 | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
| , | -- [%B24 | -- [%B24 | -- [%B24 | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
| [%B29 | 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- |
| + | 2 | 10 | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 |
| -- [%B24 | -- [%B24 | -- [%B24 | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 |
| 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 |
| , | -- [%B24 | -- [%B24 | -- [%B24 | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 | + | -- [%B24 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
| [%B25 | 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- |
| + | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 | 1 | 4 | 3 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
| [%B26 | 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- | :---:- |
| + | 2 | 10 | 1 | 9 | 7 | 8 | 1 | 7 | 1 | 6 | 1 | 5 |
| 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 |
| 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 | 54 | 27 |
The number of real solutions for the input system is:

```
#lr2
27
```

Another example of computation of real solutions illustrates the
LexTriangularPackage package constructor.

We concentrate now on the solutions with real (strictly) positive
coordinates:

\[
\begin{align*}
&lpr2 := \text{positiveSolve}(l) \text{pack} \\
&\left[\begin{array}{ccccccc}
1 & 3 & 1 & 1 & 3 & 1 & 1 \\
\%B40, & - & - & %B40 & + & - & - & %B40 & + & - & - & %B40 & + & - \\
3 & 3 & 3 & 3 & 3 & 3 & 3
\end{array}\right] \\
&\text{Type: List List RealClosure Fraction Integer}
\end{align*}
\]

Finally, we approximate the coordinates of this point with 20 exact digits:

\[
\text{[approximate}(r, 1/10**21)::\text{Float for } r \text{ in lpr2.1]}
\]
\[
\{0.32218535462608559291, 0.32218535462608559291, 0.32218535462608559291, 0.32218535462608559291\}
\]
\text{Type: List Float}

See Also:

- )show ZeroDimensionalSolvePackage

---

### ZeroDimensionalSolvePackage (ZDSOLVE)

| ZDSOLVE |
| SFRTCAT |

Exports:

- convert
- positiveSolve
- realSolve
- squareFree
- triangSolve
- univariateSolve

---

)abbrev package ZDSOLVE ZeroDimensionalSolvePackage
++ Author: Marc Moreno Maza
++ Date Created: 23/01/1999
++ Date Last Updated: 08/02/1999
++ Description:
++ A package for computing symbolically the complex and real roots of
++ zero-dimensional algebraic systems over the integer or rational
++ numbers. Complex roots are given by means of univariate representations
++ of irreducible regular chains. Real roots are given by means of tuples
++ of coordinates lying in the \spadtype{RealClosure} of the coefficient ring.
++ This constructor takes three arguments. The first one \spad{R} is the
++ coefficient ring. The second one \spad{ls} is the list of variables
++ involved in the systems to solve. The third one must be \spad{concat(ls,s)}
++ where \spad{s} is an additional symbol used for the univariate
++ representations.
++ WARNING. The third argument is not checked.
++ All operations are based on triangular decompositions.
++ The default is to compute these decompositions directly from the input
++ system by using the \spadtype{RegularChain} domain constructor.
++ The lexTriangular algorithm can also be used for computing these
++ decompositions (see \spadtype{LexTriangularPackage} package constructor).
++ For that purpose, the operations univariateSolve, realSolve and
++ positiveSolve admit an optional argument.

ZeroDimensionalSolvePackage(R,ls,ls2): Exports == Implementation where
R : Join(OrderedRing,EuclideanDomain,CharacteristicZero,RealConstant)
ls: List Symbol
ls2: List Symbol
V ==> OrderedVariableList(ls)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
P ==> Polynomial R
LP ==> List P
LS ==> List Symbol
Q ==> NewSparseMultivariatePolynomial(R,V)
U ==> SparseUnivariatePolynomial(R)
TS ==> RegularChain(R,ls)
RUR ==> Record(complexRoots: U, coordinates: LP)
K ==> Fraction R
RC ==> RealClosure(K)
PRC ==> Polynomial RC
REALSOL ==> List RC
URC ==> SparseUnivariatePolynomial RC
V2 ==> OrderedVariableList(ls2)
Q2 ==> NewSparseMultivariatePolynomial(R,V2)
E2 ==> IndexedExponents V2
ST ==> SquareFreeRegularTriangularSet(R,E2,V2,Q2)
Q2WT ==> Record(val: Q2, tower: ST)
LQ2WT ==> Record(val: List(Q2), tower: ST)
WIP ==> Record(reals: List(RC), vars: List(Symbol), pols: List(Q2))
polsetpack ==> PolynomialSetUtilitiesPackage(R,E2,V2,Q2)
normpack ==> NormalizationPackage(R,E2,V2,Q2,ST)
rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E2,V2,Q2,ST)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E2,V2,Q2,ST)
lextripack ==> LexTriangularPackage(R,ls)
Exports == with
triangSolve: (LP,B,B) -> List RegularChain(R,ls)
++ \spad{triangSolve(lp,info?,lextri?)} decomposes the variety
++ associated with \axiom{lp} into regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in the output.
++ Note that \axiom{lp} needs to generate a zero-dimensional ideal.
++ If \axiom{lp} is not zero-dimensional then the result is only
++ a decomposition of its zero-set in the sense of the closure
++ (w.r.t. Zarisky topology).
++ Moreover, if \axiom{info?} is \axiom{true} then some information is
++ displayed during the computations.
++ See zeroSetSplit from RegularTriangularSetCategory(lp,true,info?).
++ If \axiom{lextri?} is \axiom{true} then the lexTriangular algorithm
++ is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly from
++ the input
++ system by using the zeroSetSplit from RegularChain.
triangSolve: (LP,B) -> List RegularChain(R,ls)
++ \spad{triangSolve(lp)} returns the same as
++ \spad{triangSolve(lp,false)}
triangSolve: LP -> List RegularChain(R,ls)
++ \spad{triangSolve(lp)} returns the same as
++ \spad{triangSolve(lp,false,false)}
univariateSolve: RegularChain(R,ls) -> _
++ \spad{univariateSolve(ts)} returns a univariate representation
++ of \spad{ts}.
++ See rur from RationalUnivariateRepresentationPackage(lp,true).
univariateSolve: (LP,B,B,B) -> List RUR
++ \spad{univariateSolve(lp,info?,check?,lextri?)} returns a univariate
++ representation of the variety associated with \spad{lp}.
++ Moreover, if \axiom{info?} is \axiom{true} then some information is
++ displayed during the decomposition into regular chains.
++ If \axiom{check?} is \axiom{true} then the result is checked.
++ See rur from RationalUnivariateRepresentationPackage(lp,true).
++ If \axiom{lextri?} is \axiom{true} then the lexTriangular
++ algorithm is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly
++ from the input
++ system by using the zeroSetSplit from RegularChain.
univariateSolve: (LP,B,B) -> List RUR
++ \spad{univariateSolve(lp,info?,check?)} returns the same as
++ \spad{univariateSolve(lp,info?,check?,false)}.
univariateSolve: (LP,B) -> List RUR
univariateSolve: LP -> List RUR
++ \spad{univariateSolve(lp,info?)} returns the same as
++ \spad{univariateSolve(lp,info?,false,false)}.

realSolve: RegularChain(R,ls) -> List REALSOL
++ \spad{realSolve(ts)} returns the set of the points in the regular
++ zero set of \spad{ts} whose coordinates are all real.
++ The ordering of the indeterminates is reversed w.r.t. \spad{ls}.

realSolve: (LP,B,B,B) -> List REALSOL
++ \spad{realSolve(ts,info?,check?,lextri?)} returns the set of the
++ points in the variety associated with \spad{lp} whose coordinates
++ are all real.
++ Moreover, if \spad{info?} is \spad{true} then some information is
++ displayed during decomposition into regular chains.
++ If \spad{check?} is \spad{true} then the result is checked.
++ If \spad{lextri?} is \spad{true} then the lexTriangular algorithm
++ is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly from
++ the input
++ system by using the zeroSetSplit from \spadtype{RegularChain}.
++ WARNING. For each set of coordinates given by
++ \spad{realSolve(ts,info?,check?,lextri?)}
++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.

realSolve: (LP,B) -> List REALSOL
++ \spad{realSolve(ts,info?,check?)} returns the same as
++ \spad{realSolve(ts,info?,check?,false)}.

realSolve: LP -> List REALSOL
++ \spad{realSolve(lp)} returns the same as
++ \spad{realSolve(ts,false,false,false)}. 

positiveSolve: RegularChain(R,ls) -> List REALSOL
++ \spad{positiveSolve(ts)} returns the points of the regular
++ set of \spad{ts} with (real) strictly positive coordinates.

positiveSolve: (LP,B,B) -> List REALSOL
++ \spad{positiveSolve(lp,info?,lextri?)} returns the set of the points
++ in the variety associated with \spad{lp} whose coordinates are
++ (real) strictly positive.
++ Moreover, if \spad{info?} is \spad{true} then some information is
++ displayed during decomposition into regular chains.
++ If \spad{lextri?} is \spad{true} then the lexTriangular
++ algorithm is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly
++ from the input
++ system by using the zeroSetSplit from \spadtype{RegularChain}.
++ \textbf{WARNING}. For each set of coordinates given by
++ \spad{positiveSolve(lp,info?,lextri?)}
++ the ordering of the indeterminates is reversed \textit{w.r.t.} \spad{ls}.
positiveSolve: (LP,B) \to List REALSOL
++ \spad{positiveSolve(lp)} returns the same as
++ \spad{positiveSolve(lp,info?,false)}.
positiveSolve: LP \to List REALSOL
++ \spad{positiveSolve(lp)} returns the same as
++ \spad{positiveSolve(lp,false,false)}.
squareFree: (TS) \to List ST
++ \spad{squareFree(ts)} returns the square-free factorization
++ of \spad{ts}. Moreover, each factor is a Lazard triangular set
++ and the decomposition
++ is a Kalkbrener split of \spad{ts}, which is enough here for
++ the matter of solving zero-dimensional algebraic systems.
++ \textbf{WARNING}. \spad{ts} is not checked to be zero-dimensional.
convert: Q \to Q2
++ \spad{convert(q)} converts \spad{q}.
convert: P \to PRC
++ \spad{convert(p)} converts \spad{p}.
convert: Q2 \to PRC
++ \spad{convert(q)} converts \spad{q}.
convert: U \to URC
++ \spad{convert(u)} converts \spad{u}.
convert: ST \to List Q2
++ \spad{convert(st)} returns the members of \spad{st}.

Implementation == add
news: Symbol := last(ls2)$(List Symbol)
newv: V2 := (variable(news)$V2)::V2
newq: Q2 := newv :: Q2
convert(q:Q):Q2 ==
  ground? q => (ground(q))::Q2
  q2: Q2 := 0
  while not ground?(q) repeat
    v: V := mvar(q)
    d: N := mdeg(q)
    v2: V2 := (variable(convert(v)$Symbol)$V2)::V2
    iq2: Q2 := convert(init(q))@Q2
    1q2: Q2 := (v2 :: Q2)
    1q2 := 1q2 ** d
    q2 := iq2 * 1q2 + q2
    q := tail(q)
  q2 + (ground(q))::Q2

squareFree(ts:TS):List(ST) ==
  irredu?: Boolean := false
st: ST := [[newq]$(List Q2)]
lq: List(Q2) := [convert(p)$Q2 for p in parts(ts)]
lq := sort(infrittWu?,lq)
toSee: List LQ2WT := []
if irred?
    then
        lf := irreducibleFactors([first lq])$polsetpack
        lq := rest lq
        for f in lf repeat
            toSee := cons([cons(f,lq),st]$LQ2WT, toSee)
    else
        toSee := [[lq,st]$LQ2WT]
toSave: List ST := []
while not empty? toSee repeat
    lqwt := first toSee; toSee := rest toSee
    lq := lqwt.val; st := lqwt.tower
    empty? lq =>
        toSave := cons(st,toSave)
    q := first lq; lq := rest lq
    lsfqwt: List Q2WT := squareFreePart(q,st)$ST
    for sfqwt in lsfqwt repeat
        q := sfqwt.val; st := sfqwt.tower
        if not ground? init(q)
            then
                q := normalizedAssociate(q,st)$normpack
                newts := internalAugment(q,st)$ST
                newlq := [remainder(q,newts).polnum for q in lq]
                toSee := cons([newlq,newts]$LQ2WT,toSee)
        toSave

triangSolve(lp: LP, info?: B, lextri?: B): List TS ==
lq: List(Q) := [convert(p)$Q for p in lp]
lextri? => zeroSetSplit(lq,false)$lextripack
zeroSetSplit(lq,true,info?)$TS

triangSolve(lp: LP, info?: B): List TS == triangSolve(lp,info?,false)

triangSolve(lp: LP): List TS == triangSolve(lp,false)

convert(u: U): URC ==
    zero? u => 0
    ground? u => ((ground(u) :: K)::RC)::URC
    uu: URC := 0
    while not ground? u repeat
        uu := monomial((leadingCoefficient(u) :: K)::RC,degree(u)) + uu
        u := reductum u
        uu + ((ground(u) :: K)::RC)::URC

coerceFromRtoRC(r:R): RC ==
\( (r:K)::RC \)

\[
\text{convert}(p:P): PRC ==
\text{map}(\text{coerceFromRtoRC},p)$\text{PolynomialFunctions2}(R,RC)
\]

\[
\text{convert}(q2:Q2): PRC ==
p: P := \text{coerce}(q2)$Q2
\text{convert}(p)$PRC
\]

\[
\text{convert}(sts:ST): \text{List} \ Q2 ==
lq2: \text{List}(Q2) := \text{parts}(sts)$ST
lq2 := \text{sort}(\text{infRittWu?},lq2)
\text{rest}(lq2)
\]

\[
\text{realSolve}(ts: TS): \text{List} \ \text{REALSOL} ==
lsts: \text{List} \ ST := \text{squareFree}(ts)
lr: \text{REALSOL} := []
lv: \text{List} \ \text{Symbol} := []
toSee := \{[lr,lv,\text{convert}(sts)@(\text{List} \ Q2)]$\text{WIP} \text{ for} \ \text{sts} \ \text{in} \ \text{lsts}\}
toSave: \text{List} \ \text{REALSOL} := []
\text{while} \ \text{not} \ \text{empty?} \ \text{toSee} \ \text{repeat}
\text{wip} := \text{first} \ \text{toSee}; \ \text{toSee} := \text{rest} \ \text{toSee}
lr := \text{wip.reals}; \ lv := \text{wip.vars}; \ lq2 := \text{wip.pols}
(\text{empty?} \ lq2) \ \text{and} \ (\text{not} \ \text{empty?} \ lr) \Rightarrow
\text{toSave} := \text{cons}(\text{reverse}(lr),\text{toSave})
q2 := \text{first} \ lq2; \ lq2 := \text{rest} \ lq2
qrc := \text{convert}(q2)$PRC
\text{if} \ \text{not} \ \text{empty?} \ lr
\text{then}
\text{for} \ r \ \text{in} \ \text{reverse}(lr) \ \text{for} \ v \ \text{in} \ \text{reverse}(lv) \ \text{repeat}
qrc := \text{eval}(qrc,v,r)
lv := \text{cons}((\text{mainVariable}(qrc) :: \text{Symbol}),lv)
urc: URC := \text{univariate}(qrc)@URC
urcRoots := \text{allRootsOf}(urc)$RC
\text{for} \ urcRoot \ \text{in} \ urcRoots \ \text{repeat}
\text{toSee} := \text{cons}([\text{cons}(urcRoot,lr),lv,lq2]$\text{WIP}, \ \text{toSee})
\text{toSave}
\]

\[
\text{realSolve}(lp: \text{List}(P), \text{info?:Boolean, check?: Boolean, lextri?: Boolean}): \text{List} \ \text{REALSOL} ==
lts: \text{List} \ TS
lq: \text{List}(Q) := [\text{convert}(p)$Q \text{ for} \ p \ \text{in} \ lp]
\text{if} \ \text{lextri?}
\text{then}
lts := \text{zeroSetSplit}(lq,\text{false})$\text{lextrpack}
\text{else}
lts := \text{zeroSetSplit}(lq,\text{true},\text{info?})$\text{TS}
lsts: \text{List} \ ST := []
\text{for} \ ts \ \text{in} \ lts \ \text{repeat}
lsts := \text{concat}(\text{squareFree}(ts), \ \text{lsts})
lsts := \text{removeSuperfluousQuasiComponents}(lsts)$\text{quasicomppack}
lr: REALSOL := []
lv: List Symbol := []
toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
toSave: List REALSOL := []
while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    lr := wip.reals; lv := wip.vars; lq2 := wip.pols
    (empty? lq2) and (not empty? lr) =>
        toSave := cons(reverse(lr), toSave)
    q2 := first lq2; lq2 := rest lq2
    qrc := convert(q2)@PRC
    if not empty? lr
        then
            for r in reverse(lr) for v in reverse(lv) repeat
                qrc := eval(qrc, v, r)
            lv := cons((mainVariable(qrc) :: Symbol), lv)
    urc := univariate(qrc)@URC
    urcRoots := allRootsOf(urc)$RC
    for urcRoot in urcRoots repeat
        toSee := cons([cons(urcRoot, lr), lv, lq2]$WIP, toSee)
    if check?
        then
            for p in lp repeat
                for realsol in toSave repeat
                    prc := convert(p)@PRC
                for rr in realsol for symb in reverse(ls) repeat
                    prc := eval(p, symb, rr)
                not zero? prc =>
                    error "realSolve$ZDSOLVE: bad result"
    toSave

realSolve(lp: List(P), info?: Boolean, check?: Boolean): List REALSOL ==
    realSolve(lp, info?, check?, false)

realSolve(lp: List(P), info?: Boolean): List REALSOL ==
    realSolve(lp, info?, false, false)

realSolve(lp: List(P)): List REALSOL ==
    realSolve(lp, false, false, false)

positiveSolve(ts: TS): List REALSOL ==
lsts: List ST := squareFree(ts)
lr: REALSOL := []
lv: List Symbol := []
toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
toSave: List REALSOL := []
while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    lr := wip.reals; lv := wip.vars; lq2 := wip.pols
    (empty? lq2) and (not empty? lr) =>
toSave := cons(reverse(lr),toSave)
q2 := first lq2; lq2 := rest lq2
qrc := convert(q2)@PRC
if not empty? lr
  then
    for r in reverse(lr) for v in reverse(lv) repeat
      qrc := eval(qrc,v,r)
lv := cons((mainVariable(qrc) :: Symbol),lv)
urc: URC := univariate(qrc)@URC
urcRoots := allRootsOf(urc)$RC
for urcRoot in urcRoots repeat
  if positive? urcRoot
    then
      toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
toSave

positiveSolve(lp: List(P), info?:Boolean, lextri?: Boolean): List REALSOL ==
lts: List TS
lq: List(Q) := [convert(p)$Q for p in lp]
if lextri?
  then
    lts := zeroSetSplit(lq,false)$lextripack
  else
    lts := zeroSetSplit(lq,true,info?)$TS
lsts: List ST := []
for ts in lts repeat
  lsts := concat(squareFree(ts), lsts)
lsts := removeSuperfluousQuasiComponents(lsts)$quasicomppack
lr: REALSOL := []
lv: List Symbol := []
toSee := [[lr,lv,convert(sts)@(List Q2)$WIP for sts in lsts]
toSave: List REALSOL := []
while not empty? toSee repeat
  wip := first toSee; toSee := rest toSee
  lr := wip.reals; lv := wip.vars; lq2 := wip.pols
  (empty? lq2) and (not empty? lr) =>
    toSave := cons(reverse(lr),toSave)
q2 := first lq2; lq2 := rest lq2
qrc := convert(q2)@PRC
if not empty? lr
  then
    for r in reverse(lr) for v in reverse(lv) repeat
      qrc := eval(qrc,v,r)
lv := cons((mainVariable(qrc) :: Symbol),lv)
urc: URC := univariate(qrc)@URC
urcRoots := allRootsOf(urc)$RC
for urcRoot in urcRoots repeat
  if positive? urcRoot
    then
      toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
toSave

positiveSolve(lp: List(P), info?: Boolean): List REALSOL ==
  positiveSolve(lp, info?, false)

positiveSolve(lp: List(P)): List REALSOL ==
  positiveSolve(lp, false, false)

univariateSolve(ts: TS): List RUR ==
  toSee: List ST := squareFree(ts)
  toSave: List RUR := []
  for st in toSee repeat
    lus: List ST := rur(st,true)$rurpack
    for us in lus repeat
      g: U := univariate(select(us,newv)::Q2)$Q2
      lc: LP := [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
      toSave := cons([g,lc]$RUR, toSave)
  toSave

univariateSolve(lp: List(P), info?: Boolean, check?: Boolean, lextri?: Boolean): List RUR ==
  lts: List TS
  lq: List(Q) := [convert(p)$Q for p in lp]
  if lextri?
    then
      lts := zeroSetSplit(lq,false)$lextripack
    else
      lts := zeroSetSplit(lq,true,info?)$TS
  toSee: List ST := []
  for ts in lts repeat
    toSee := concat(squareFree(ts), toSee)
  toSee := removeSuperfluousQuasiComponents(toSee)$quasicomppack
  toSave: List RUR := []
  if check?
    then
      lq2: List(Q2) := [convert(p)$Q2 for p in lp]
      for st in toSee repeat
        lus: List ST := rur(st,true)$rurpack
        for us in lus repeat
          if check?
            then
              rems: List(Q2) := [removeZero(q2,us)$ST for q2 in lq2]
              not every?(zero?,rems) =>
                output(st::OutputForm)$OutputPackage
                output("Has a bad RUR component:")$OutputPackage
                output(us::OutputForm)$OutputPackage
                error "univariateSolve$ZDSOLVE: bad RUR"
              g: U := univariate(select(us,newv)::Q2)$Q2
              lc: LP := _
                [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
              toSave := cons([g,lc]$RUR, toSave)
toSave

univariateSolve(lp: List(P), info?:Boolean, check?:Boolean): List RUR ==
  univariateSolve(lp,info?,check?,false)

univariateSolve(lp: List(P), info?:Boolean): List RUR ==
  univariateSolve(lp,info?,false,false)

univariateSolve(lp: List(P)): List RUR ==
  univariateSolve(lp,false,false,false)

— ZDSOLVE.dotabb —

"ZDSOLVE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ZDSOLVE"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"ZDSOLVE" -> "SFRTCAT"

—
Chapter 28

Chunk collections

— algebra —

\getchunk{package AFALGRO AffineAlgebraicSetComputeWithGroebnerBasis}
\getchunk{package AFALGRES AffineAlgebraicSetComputeWithResultant}
\getchunk{package AF AlgebraicFunction}
\getchunk{package INTHERAL AlgebraicHermiteIntegration}
\getchunk{package INTEGR AlgebraicIntegrate}
\getchunk{package INTAF AlgebraicIntegration}
\getchunk{package ALGMANIP AlgebraicManipulations}
\getchunk{package ALGMFACT AlgebraicMultFact}
\getchunk{package ALGPKG AlgebraPackage}
\getchunk{package ALGFACT AlgFactor}
\getchunk{package INTPACK AnnaNumericalIntegrationPackage}
\getchunk{package OPTPACK AnnaNumericalOptimizationPackage}
\getchunk{package ODEPACK AnnaOrdinaryDifferentialEquationPackage}
\getchunk{package PDEPACK AnnaPartialDifferentialEquationPackage}
\getchunk{package ANY1 AnyFunctions1}
\getchunk{package APPRULE ApplyRules}
\getchunk{package APPLYORE ApplyUnivariateSkewPolynomial}
\getchunk{package ASSOCEQ AssociatedEquations}
\getchunk{package PMPRED AttachPredicates}
\getchunk{package AXSERV AxiomServer}

\getchunk{package BALFACT BalancedFactorisation}
\getchunk{package BOP1 BasicOperatorFunctions1}
\getchunk{package BEZOUT BezoutMatrix}
\getchunk{package BLUPPACK BlowUpPackage}
\getchunk{package BOUNDZRO BoundIntegerRoots}
\getchunk{package BRILL BrillhartTests}

\getchunk{package CARTEM2 CartesianTensorFunctions2}
CHAPTER 28. CHUNK COLLECTIONS

\getchunk{package CHVAR ChangeOfVariable}
\getchunk{package CPIMA CharacteristicPolynomialInMonogenousAlgebra}
\getchunk{package CHARPOL CharacteristicPolynomialPackage}
\getchunk{package IBACHIN ChineseRemainderToolsForIntegralBases}
\getchunk{package CVMP CoerceVectorMatrixPackage}
\getchunk{package COMBF CombinatorialFunction}
\getchunk{package CDEN CommonDenominator}
\getchunk{package COMMONOP CommonOperators}
\getchunk{package COMMPC CommuteUnivariatePolynomialCategory}
\getchunk{package COMPFAC ComplexFactorization}
\getchunk{package COMPLEX2 ComplexFunctions2}
\getchunk{package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation}
\getchunk{package COMPLAT ComplexPattern}
\getchunk{package CPMATCH ComplexPatternMatch}
\getchunk{package CRFP ComplexRootFindingPackage}
\getchunk{package CMPLXRT ComplexRootPackage}
\getchunk{package CTRIGMNP ComplexTrigonometricManipulations}
\getchunk{package D01AGNT d01AgentsPackage}
\getchunk{package D01WGTS d01WeightsPackage}
\getchunk{package D02AGNT d02AgentsPackage}
\getchunk{package D03AGNT d03AgentsPackage}
\getchunk{package DEGRED DegreeReductionPackage}
\getchunk{package DTP DesingTreePackage}
\getchunk{package DIOSP DiophantineSolutionPackage}
\getchunk{package DIRPROD2 DirectProductFunctions2}
\getchunk{package DLP DiscreteLogarithmPackage}
\getchunk{package DISPLAY DisplayPackage}
\getchunk{package DDFACT DistinctDegreeFactorize}
\getchunk{package DFSFUN DoubleFloatSpecialFunctions}
\getchunk{package DBLRESP DoubleResultantPackage}
\getchunk{package DRAWCX DrawComplex}
\getchunk{package DRAWHACK DrawNumericHack}
\getchunk{package DROPT0 DrawOptionFunctions0}
\getchunk{package DROPT1 DrawOptionFunctions1}
\getchunk{package DO1AGNT d01AgentsPackage}
\getchunk{package DO1WGS d01WeightsPackage}
\getchunk{package DO2AGNT d02AgentsPackage}
\getchunk{package DO3AGNT d03AgentsPackage}
\getchunk{package DEFINTEF ElementaryFunctionDefiniteIntegration}
\getchunk{package LODEEF ElementaryFunctionLODESolver}
\getchunk{package ODEEF ElementaryFunctionODESolver}
\getchunk{package SIGNEF ElementaryFunctionSign}
\getchunk{package EFSTRUC ElementaryFunctionStructurePackage}
\getchunk{package INTEF ElementaryIntegration}
\getchunk{package RDEEF ElementaryRischDE}
\getchunk{package RDEEFS ElementaryRischDESystem}
\getchunk{package ELFUTS EllipticFunctionsUnivariateTaylorSeries}
\getchunk{package EQ2 EquationFunctions2}
\getchunk{package ERROR ErrorFunctions}
\getchunk{package GBEUCLID EuclideanGroebnerBasisPackage}
\getchunk{package EVALCYC EvaluateCycleIndicators}
\getchunk{package ESCONT ExpertSystemContinuityPackage}
\getchunk{package ESCONT1 ExpertSystemContinuityPackage1}
\getchunk{package ESTOOLS ExpertSystemToolsPackage}
\getchunk{package ESTOOLS1 ExpertSystemToolsPackage1}
\getchunk{package ESTOOLS2 ExpertSystemToolsPackage2}
\getchunk{package EXPR2 ExpressionFunctions2}
\getchunk{package EXPRSOL ExpressionSolve}
\getchunk{package ES1 ExpressionSpaceFunctions1}
\getchunk{package ES2 ExpressionSpaceFunctions2}
\getchunk{package EXPRODE ExpressionSpaceODESolver}
\getchunk{package OMEXPR ExpressionToOpenMath}
\getchunk{package EXPR2UPS ExpressionToUnivariatePowerSeries}
\getchunk{package EXPRTUBE ExpressionTubePlot}
\getchunk{package EXP3D Export3D}
\getchunk{package E04AGNT e04AgentsPackage}
\getchunk{package FACTFUNC FactoredFunctions}
\getchunk{package FR2 FactoredFunctions2}
\getchunk{package FRUTIL FactoredFunctionUtilities}
\getchunk{package FACUTIL FactoringUtilities}
\getchunk{package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber}
\getchunk{package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber}
\getchunk{package FGLMICPK FGLMIfCanPackage}
\getchunk{package FORDER FindOrderFinite}
\getchunk{package FAMR2 FiniteAbelianMonoidRingFunctions2}
\getchunk{package FDIV2 FiniteDivisorFunctions2}
\getchunk{package FFACTOR FiniteFieldFactorization}
\getchunk{package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect}
\getchunk{package FFF FiniteFieldFunctions}
\getchunk{package FFHOM FiniteFieldHomomorphisms}
\getchunk{package FFPOLY FiniteFieldPolynomialPackage}
\getchunk{package FFPOLY2 FiniteFieldPolynomialPackage2}
\getchunk{package FFSLPE FiniteFieldSolveLinearPolynomialEquation}
\getchunk{package FFSQFR FiniteFieldSquareFreeDecomposition}
\getchunk{package FLAGG2 FiniteLinearAggregateFunctions2}
\getchunk{package FASORT FiniteLinearAggregateSort}
\getchunk{package FSAGG2 FiniteSetAggregateFunctions2}
\getchunk{package FLOATCP FloatingComplexPackage}
\getchunk{package FLOATRP FloatingRealPackage}
\getchunk{package FCPAK1 FortranCodePackage1}
\getchunk{package FOP FortranOutputStackPackage}
\getchunk{package FORT FortranPackage}
CHAPTER 28. CHUNK COLLECTIONS

\getchunk{package FRIDEAL2 FractionalIdealFunctions2}
\getchunk{package FFFG FractionFreeFastGaussian}
\getchunk{package FFFGF FractionFreeFastGaussianFractions}
\getchunk{package FRAC2 FractionFunctions2}
\getchunk{package FRNAAF2 FramedNonAssociativeAlgebraFunctions2}
\getchunk{package FFSPECF FunctionalSpecialFunction}
\getchunk{package FFCAT2 FunctionFieldCategoryFunctions2}
\getchunk{package FFINTBAS FunctionFieldIntegralBasis}
\getchunk{package PMASSFS FunctionSpaceAssertions}
\getchunk{package PMPREDFS FunctionSpaceAttachPredicates}
\getchunk{package FSCINT FunctionSpaceComplexIntegration}
\getchunk{package FS2 FunctionSpaceFunctions2}
\getchunk{package FS2RED FunctionSpaceReduce}
\getchunk{package SUMFS FunctionSpaceSum}
\getchunk{package FS2EXPXP FunctionSpaceToExponentialExpansion}
\getchunk{package FS2UPS FunctionSpaceToUnivariatePowerSeries}
\getchunk{package FSUPFACT FunctionSpaceUnivariatePolynomialFactor}
\getchunk{package GALFACTU GaloisGroupFactorizationUtilities}
\getchunk{package GALFACT GaloisGroupFactorizer}
\getchunk{package GALPOLYU GaloisGroupPolynomialUtilities}
\getchunk{package GALUTIL GaloisGroupUtilities}
\getchunk{package GAUSSFAC GaussianFactorizationPackage}
\getchunk{package GENHENSEL GeneralHenselPackage}
\getchunk{package GENNFAC GeneralizedMultivariateFactorizer}
\getchunk{package GPAFF GeneralPackageForAlgebraicFunctionField}
\getchunk{package GENPGCD GeneralPolynomialGcdPackage}
\getchunk{package GENUPS GeneralUnivariatePowerSeries}
\getchunk{package GENEXZ GenExEuclid}
\getchunk{package GENUFAC GenUFactorize}
\getchunk{package INTGO GenusZeroIntegration}
\getchunk{package GDEPER GosperSummationMethod}
\getchunk{package GRDEF GraphicsDefaults}
\getchunk{package GRAY GrayCode}
\getchunk{package GBF GroebnerFactorizationPackage}
\getchunk{package GBINTERN GroebnerInternalPackage}
\getchunk{package GB GroebnerPackage}
\getchunk{package GROEBSOL GroebnerSolve}
\getchunk{package GUESS Guess}
\getchunk{package GUESSAN GuessAlgebraicNumber}
\getchunk{package GUESSF GuessFinite}
\getchunk{package GUESSF1 GuessFiniteFunctions}
\getchunk{package GUESSINT GuessInteger}
\getchunk{package GUESSP GuessPolynomial}
\getchunk{package GUESSUP GuessUnivariatePolynomial}
\getchunk{package HB HallBasis}
\getchunk{package HEUGCD HeuGcd}
\getchunk{package IDECOMP IdealDecompositionPackage}
\getchunk{package INCRMAPS IncrementingMaps}
\getchunk{package INPRDDO InfiniteProductCharacteristicZero}
\getchunk{package INPRODFP InfiniteProductPrimeField}
\getchunk{package ITFUN2 InfiniteTupleFunctions2}
\getchunk{package ITFUN3 InfiniteTupleFunctions3}
\getchunk{package INFINITY Infinity}
\getchunk{package IALGFCT InnerAlgFactor}
\getchunk{package ICDEN InnerCommonDenominator}
\getchunk{package IMATLIN InnerMatrixLinearAlgebraFunctions}
\getchunk{package IMATQF InnerMatrixQuotientFieldFunctions}
\getchunk{package INMODGCD InnerModularGcd}
\getchunk{package INMFACT InnerMultFact}
\getchunk{package INBFF InnerNormalBasisFieldFunctions}
\getchunk{package INEP InnerNumericEigenPackage}
\getchunk{package INFSF InnerNumericFloatSolvePackage}
\getchunk{package INPSIGN InnerPolySign}
\getchunk{package ITRIGMNP InnerTrigonometricManipulations}
\getchunk{package INFORM1 InputFormFunctions1}
\getchunk{package INTBIT IntegerBits}
\getchunk{package COMBINAT IntegerCombinatoricFunctions}
\getchunk{package INTFACT IntegerFactorizationPackage}
\getchunk{package ZLINDEP IntegerLinearDependence}
\getchunk{package INTHET Theory IntegerNumberTheoryFunctions}
\getchunk{package PRIMES IntegerPrimesPackage}
\getchunk{package INTRET IntegerRetractions}
\getchunk{package IROOT IntegerRoots}
\getchunk{package INTSLPE IntegerSolveLinearPolynomialEquation}
\getchunk{package IBAITGL IntegralBasisTools}
\getchunk{package IBPTOOLS IntegralBasisPolynomialTools}
\getchunk{package IR2 IntegrationResultFunctions2}
\getchunk{package IR2F IntegrationResultToFunction}
\getchunk{package IRRF2F IntegrationResultRFToFunction}
\getchunk{package INTTOOLS IntegrationTools}
\getchunk{package INTERGB InterfaceGroebnerPackage}
\getchunk{package IPRNTPK InternalPrintPackage}
\getchunk{package IRRURPK InternalRationalUnivariateRepresentationPackage}
\getchunk{package INFRSP InterpolateFormsPackage}
\getchunk{package INTDIVP IntersectionDivisorPackage}
\getchunk{package IRRREDFFX IrredPolyOverFiniteField}
\getchunk{package IRSN IrrRepSymNatPackage}
\getchunk{package InVlAPlA InverseLaplaceTransform}
\getchunk{package KERNEL2 KernelFunctions2}
\getchunk{package KOVACIC Kovacic}
\getchunk{package LAPLACE LaplaceTransform}
\getchunk{package LAZM3PK LazardSetSolvingPackage}
\getchunk{package LEADCDET LeadingCoefDetermination}
\getchunk{package LEXTRIPK LexTriangularPackage}
\getchunk{package LINDEP LinearDependence}
\getchunk{package LODDOF LinearOrdinaryDifferentialOperatorFactorizer}
\getchunk{package LODOOPS LinearOrdinaryDifferentialOperatorsOps}
\getchunk{package LPEFRAC LinearPolynomialEquationByFractions}
\getchunk{package LSMP LinearSystemMatrixPackage}
\getchunk{package LSMP1 LinearSystemMatrixPackage1}
\getchunk{package LSPP LinearSystemPolynomialPackage}
\getchunk{package LGRGBP LinGroebnerPackage}
\getchunk{package LOP LinesOpPack}
\getchunk{package LF LiouvillianFunction}
\getchunk{package LIST2 ListFunctions2}
\getchunk{package LIST3 ListFunctions3}
\getchunk{package LIST2MAP ListToMap}
\getchunk{package LPARSPT LocalParametrizationOfSimplePointPackage}
\getchunk{package MKBCFUNC MakeBinaryCompiledFunction}
\getchunk{package MKFLCFN MakeFloatCompiledFunction}
\getchunk{package MKFUNC MakeFunction}
\getchunk{package MKRECORD MakeRecord}
\getchunk{package MKUCFUNC MakeUnaryCompiledFunction}
\getchunk{package MAPHACK1 MappingPackageInternalHacks1}
\getchunk{package MAPHACK2 MappingPackageInternalHacks2}
\getchunk{package MAPHACK3 MappingPackageInternalHacks3}
\getchunk{package MAPPKG1 MappingPackage1}
\getchunk{package MAPPKG2 MappingPackage2}
\getchunk{package MAPPKG3 MappingPackage3}
\getchunk{package MAPPKG4 MappingPackage4}
\getchunk{package MATCAT2 MatrixCategoryFunctions2}
\getchunk{package MCDEN MatrixCommonDenominator}
\getchunk{package MATLIN MatrixLinearAlgebraFunctions}
\getchunk{package MAMA MatrixManipulation}
\getchunk{package MTHING MergeThing}
\getchunk{package MESH MeshCreationRoutinesForThreeDimensions}
\getchunk{package MDDFACT ModularDistinctDegreeFactorizer}
\getchunk{package MHRWRED ModularHermitianRowReduction}
\getchunk{package MRF2 MonoidRingFunctions2}
\getchunk{package MONUTOOL MonomialExtensionTools}
\getchunk{package MSYSCMD MoreSystemCommands}
\getchunk{package MPCPF MPolyCatPolyFactorizer}
\getchunk{package MPRFF MPolyCatRationalFunctionFactorizer}
\getchunk{package MPC2 MPolyCatFunctions2}
\getchunk{package MPC3 MPolyCatFunctions3}
\getchunk{package MRATFAC MRationalFactorize}
\getchunk{package MFINFAC MultFiniteFactorize}
\getchunk{package MMAP MultipleMap}
\getchunk{package MCALCFN MultiVariableCalculusFunctions}
\getchunk{package MULTFACT MultivariateFactorize}
\getchunk{package MLIFT MultivariateLifting}  
\getchunk{package MULTSQFR MultivariateSquareFree}  
\getchunk{package NAGF02 NagEigenPackage}  
\getchunk{package NAGE02 NagFittingPackage}  
\getchunk{package NAGF04 NagLinearEquationSolvingPackage}  
\getchunk{package NAGSP NagLinkSupportPackage}  
\getchunk{package NAGD01 NagIntegrationPackage}  
\getchunk{package NAGE01 NagInterpolationPackage}  
\getchunk{package NAGFO7 NagLapack}  
\getchunk{package NAGFO1 NagMatrixOperationsPackage}  
\getchunk{package NAGE04 NagOptimisationPackage}  
\getchunk{package NAGD02 NagOrdinaryDifferentialEquationsPackage}  
\getchunk{package NAGD03 NagPartialDifferentialEquationsPackage}  
\getchunk{package NAGCO2 NagPolynomialRootsPackage}  
\getchunk{package NAGCO5 NagRootFindingPackage}  
\getchunk{package NAGC06 NagSeriesSummationPackage}  
\getchunk{package NAGS NagSpecialFunctionsPackage}  
\getchunk{package NSUP2 NewSparseUnivariatePolynomialFunctions2}  
\getchunk{package NEWTON NewtonInterpolation}  
\getchunk{package NPOLYGON NewtonPolygon}  
\getchunk{package NCODIV NonCommutativeOperatorDivision}  
\getchunk{package NONE1 NoneFunctions1}  
\getchunk{package NODE1 NonLinearFirstOrderODESolver}  
\getchunk{package NLINSOL NonLinearSolvePackage}  
\getchunk{package NORMPK NormalizationPackage}  
\getchunk{package NORMMA NormInMonogenicAlgebra}  
\getchunk{package NORMRET NormRetractPackage}  
\getchunk{package NPCOEF NPCoef}  
\getchunk{package NFINTBAS NumberFieldIntegralBasis}  
\getchunk{package NUMFMT NumberFormats}  
\getchunk{package NTPOLFW NumberTheoreticPolynomialFunctions}  
\getchunk{package NUMERIC Numeric}  
\getchunk{package NUMODE NumericalOrdinaryDifferentialEquations}  
\getchunk{package NUMQUAD NumericalQuadrature}  
\getchunk{package NCEP NumericComplexEigenPackage}  
\getchunk{package NCFTRAC NumericContinuedFraction}  
\getchunk{package NREP NumericRealEigenPackage}  
\getchunk{package NUMTUBE NumericTubePlot}  
\getchunk{package OCTCT2 OctonionCategoryFunctions2}  
\getchunk{package ODEINT ODEIntegration}  
\getchunk{package ODETOOLS ODETools}  
\getchunk{package ARRAY12 OneDimensionalArrayFunctions2}  
\getchunk{package ONECOMP2 OnePointCompletionFunctions2}  
\getchunk{package OMPKG OpenMathPackage}  
\getchunk{package OMSERVER OpenMathServerPackage}  
\getchunk{package OPQUERY OperationsQuery}  
\getchunk{package ORDCOMP2 OrderedCompletionFunctions2}  
\getchunk{package ORDFUNS OrderingFunctions}
\getchunk{package ODERTRIC RationalRicDE}
\getchunk{package RURPK RationalUnivariateRepresentationPackage}
\getchunk{package POLUTIL RealPolynomialUtilitiesPackage}
\getchunk{package REALSOLV RealSolvePackage}
\getchunk{package REAL0 RealZeroPackage}
\getchunk{package REAL0Q RealZeroPackageQ}
\getchunk{package RMCAT2 RectangularMatrixCategoryFunctions2}
\getchunk{package RECP RecurrenceOperator}
\getchunk{package RDIV ReducedDivisor}
\getchunk{package ODERED ReduceLODE}
\getchunk{package REDORDER ReductionOfOrder}
\getchunk{package RSDCMPK RegularSetDecompositionPackage}
\getchunk{package RSETGCD RegularTriangularSetGcdPackage}
\getchunk{package REPDB RepeatedDoubling}
\getchunk{package REPSQ RepeatedSquaring}
\getchunk{package REP1 RepresentationPackage1}
\getchunk{package REP2 RepresentationPackage2}
\getchunk{package RESLATC ResolveLatticeCompletion}
\getchunk{package RETSOL RetractSolvePackage}
\getchunk{package RFP RootsFindingPackage}
\getchunk{package SAERFFC SAERationalFunctionAlgFactor}
\getchunk{package FORMULA1 ScriptFormulaFormat1}
\getchunk{package SEGBIND2 SegmentBindingFunctions2}
\getchunk{package SEG2 SegmentFunctions2}
\getchunk{package SAEFACT SimpleAlgebraicExtensionAlgFactor}
\getchunk{package SIMPAN SimplifyAlgebraicNumberConvertPackage}
\getchunk{package SMITH SmithNormalForm}
\getchunk{package SCACHE SortedCache}
\getchunk{package SORTPAK SortPackage}
\getchunk{package SUP2 SparseUnivariatePolynomialFunctions2}
\getchunk{package SPECOUT SpecialOutputPackage}
\getchunk{package SFQMPK SquareFreeQuasiComponentPackage}
\getchunk{package SRDCMPK SquareFreeRegularSetDecompositionPackage}
\getchunk{package SFRGCD SquareFreeRegularTriangularSetGcdPackage}
\getchunk{package MATSTOR StorageEfficientMatrixOperations}
\getchunk{package STREAM1 StreamFunctions1}
\getchunk{package STREAM2 StreamFunctions2}
\getchunk{package STREAM3 StreamFunctions3}
\getchunk{package STINPROD StreamInfiniteProduct}
\getchunk{package STTAYLOR StreamTaylorSeriesOperations}
\getchunk{package STNSR StreamTensor}
\getchunk{package STTF StreamTranscendentalFunctions}
\getchunk{package STTFNC StreamTranscendentalFunctionsNonCommutative}
\getchunk{package SCPKG StructuralConstantsPackage}
\getchunk{package SHP SturmHabichtPackage}
\getchunk{package SUBRESP SubResultantPackage}
\getchunk{package SUPFRACF SupFractionFactorizer}
\getchunk{package ODESYS SystemODESolver}
\getchunk{package SYSSOLP SystemSolvePackage}

CHAPTER 28. CHUNK COLLECTIONS
\getchunk{package SGCF SymmetricGroupCombinatoricFunctions}
\getchunk{package SYMFUNC SymmetricFunctions}
\getchunk{package TABLBUMP TableauxBumpers}
\getchunk{package TBCMPPK TabulatedComputationPackage}
\getchunk{package TANEXP TangentExpansions}
\getchunk{package UTSSOL TaylorSolve}
\getchunk{package TEMUTL TemplateUtilities}
\getchunk{package TEX1 TexFormat1}
\getchunk{package TOOLSIGN ToolsForSign}
\getchunk{package DRAW TopLevelDrawFunctions}
\getchunk{package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves}
\getchunk{package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions}
\getchunk{package DRAWPT TopLevelDrawFunctionsForPoints}
\getchunk{package TOPSP TopLevelThreeSpace}
\getchunk{package INTHERTR TranscendentalHermiteIntegration}
\getchunk{package INTR TranscendentalIntegration}
\getchunk{package TRMANIP TranscendentalManipulations}
\getchunk{package RDET TranscendentalRischDE}
\getchunk{package RDETRS TranscendentalRischDESystem}
\getchunk{package SOLVETRA TransSolvePackage}
\getchunk{package SOLVESER TransSolvePackageService}
\getchunk{package TRIMAT TriangularMatrixOperations}
\getchunk{package TRIGMNIP TrigonometricManipulations}
\getchunk{package TUBETOOL TubePlotTools}
\getchunk{package CLIP TwoDimensionalPlotClipping}
\getchunk{package TWOFACT TwoFactorize}
\getchunk{package UNIFACT UnivariateFactorize}
\getchunk{package UPS1 UnivariateFormalPowerSeriesFunctions}
\getchunk{package ULS2 UnivariateLaurentSeriesFunctions2}
\getchunk{package UPOLY2 UnivariatePolynomialCategoryFunctions2}
\getchunk{package UPCDEN UnivariatePolynomialCommonDenominator}
\getchunk{package UPDECOMP UnivariatePolynomialDecompositionPackage}
\getchunk{package UPDIVP UnivariatePolynomialDivisionPackage}
\getchunk{package UP2 UnivariatePolynomialFunctions2}
\getchunk{package UPM2 UnivariatePolynomialMultiplicationPackage}
\getchunk{package UPSQFREE UnivariatePolynomialSquareFree}
\getchunk{package UPSX2 UnivariatePuiseuxSeriesFunctions2}
\getchunk{package OREPCTU UnivariateSkewPolynomialCategoryOps}
\getchunk{package UTS2 UnivariateTaylorSeriesFunctions2}
\getchunk{package UTSODE UnivariateTaylorSeriesODESolver}
\getchunk{package UNISEG2 UniversalSegmentFunctions2}
\getchunk{package UDPO UserDefinedPartialOrdering}
\getchunk{package UDVO UserDefinedVariableOrdering}
\getchunk{package UTSDETL UTSadetools}
\getchunk{package VECTOR2 VectorFunctions2}
\getchunk{package VIEWDEF ViewDefaultsPackage}
\getchunk{package VIEW ViewportPackage}
\getchunk{package WEIER WeierstrassPreparation}
\getchunk{package WFFINTBS WildFunctionFieldIntegralBasis}
\getchunk{package XEXPPKG XExponentialPackage}
\getchunk{package ZDSOLVE ZeroDimensionalSolvePackage}
Chapter 29

Index
Index

-?
  STTAYLOR, 3437

???
  AF, 14
  COMBF, 198
  INBFF, 1130
  MAPPKG1, 1484
  MATSTOR, 3422
  STTF, 3449
  STTFNC, 3460

???
  INBFF, 1130
  MAPPKG3, 1505
  MAPPKG4, 1512
  PRS, 3117
  STTAYLOR, 3437
  TUBETOOL, 3631

?+?
  MAPPKG4, 1512
  STTAYLOR, 3437
  TUBETOOL, 3631

?~?
  MAPPKG4, 1512
  STTAYLOR, 3437
  TUBETOOL, 3631

?/?
  INBFF, 1130
  MAPPKG4, 1512
  STTAYLOR, 3437

abelianGroup
  PGE, 2945

abs
  FSPECF, 775

accuracyIF
  D02AGNT, 420
aColumn

MAMA, 1543
acos
  EF, 450
  STTF, 3449
  STTFNC, 3460
acosh
  EF, 450
acot
  EF, 450
  STTF, 3449
  STTFNC, 3460
acoth
  EF, 450
acsc
  EF, 450
  STTF, 3449
  STTFNC, 3460
acsch
  EF, 450
aCubic
  SOLVEFOR, 3065
adaptive
  DROPT0, 399
  GRDEF, 948
addBadValue
  PATTERN1, 2892
adddiag
  STTAYLOR, 3437
adjoint
  MATLIN, 1520
adjunctionDivisor
  DTP, 303
  GPAFF, 889
  PAFF, 2821
  PAFFFF, 2829
AF, 14
  ???, 14

3836
belong?, 14
definingPolynomial, 14
droot, 14
inrootof, 14
irroot, 14
minPoly, 14
operator, 14
rootOf, 14
AFALGGRO, 4
    affineAlgSet, 4
    affineRationalPoints, 4
    affineSingularPoints, 4
AFALGRES, 9
    affineAlgSet, 9
    affineAlgSetLocal, 9
    affineRationalPoints, 9
    affineSingularPoints, 9
    allPairsAmong, 9
    polyRing2UPUP, 9
AffineAlgebraicSetComputeWithGroebnerBasis, 4
    ALGMANIP, 34
    ratDenom, 34
    ratPoly, 34
    rootKerSimp, 34
    rootPower, 34
    rootProduct, 34
    rootSimp, 34
    rootSplit, 34
    ALGMFACT, 39
    factor, 39
    ALGPKG, 42
    basis, 42
    basisOfCenter, 42
    basisOfCentroid, 42
    basisOfCommutingElements, 42
    basisOfLeftAnnihilator, 42
    basisOfLeftNucleus, 42
    basisOfLeftNucloid, 42
    basisOfMiddleNucleus, 42
    basisOfNucleus, 42
    basisOfRightAnnihilator, 42
    basisOfRightNucleus, 42
    basisOfRightNucloid, 42
    ODEPAL, 3146
    algebraicDecompose
        RSDCMPK, 3310
        SRDCMPK, 3404
    AlgebraicFunction, 14
    AlgebraicHermiteIntegration, 19
    AlgebraicIntegrate, 23
    AlgebraicIntegration, 31
    AlgebraicManipulations, 34
    AlgebraicMultFact, 39
    algebraicSet
        PRJALGPK, 2961
        algebraicSort
            QCMPACK, 3156
            SFQCMPK, 3394
            AlgebraPackage, 42
            ALGFACT, 52
doublyTransitive?, 52
    factor, 52
    split, 52
    AlgFactor, 52
    algint
        INTAF, 31
    algintegrate
        INTALG, 23
rightRank, 42
weakBiRank, 42
aLinear
SOLVEFOR, 3065
allPairsAmong
AFALGRES, 9
alternating
CYCLES, 281
alternatingGroup
PGE, 2945
anfactor
FSUPFACT, 843
AnnaNumericalIntegrationPackage, 56
AnnaNumericalOptimizationPackage, 67
AnnaOrdinaryDifferentialEquationPackage,
AnnaPartialDifferentialEquationPackage, 86
anticoord
LGROBP, 1420
antisymmetricTensors
REP1, 3330
ANY1, 93
coerce, 93
retract, 93
retractable?, 93
retractIfCan, 93
AnyFunctions1, 93
API, 101
ApplicationProgramInterface, 101
apply
APPLYORE, 108
OREPCTO, 3684
APPLYORE, 108
apply, 108
ApplyRules, 104
applyRules
APPRULE, 104
applyTransform
BLUPPACK, 164
ApplyUnivariateSkewPolynomial, 108
approxNthRoot
IROOT, 1216
approxSqrt
IROOT, 1216
APPRULE, 104
applyRules, 104
localUnquote, 104
aQuadratic
SOLVEFOR, 3065
aQuartic
SOLVEFOR, 3065
areEquivalent?
REP2, 3338
arg1
MAPHACK2, 1472
arg2
MAPHACK2, 1472
aromberg
NUMQUAD, 2762
aRow
MAMA, 1543
array
ARRAY12, 2796
map, 2796
reduce, 2796
scan, 2796
asec
EF, 450
STTF, 3449
STTFNC, 3460
asech
EF, 450
asimpson
NUMQUAD, 2762
asin
EF, 450
STTF, 3449
STTFNC, 3460
asinh
EF, 450
aspFilename
NAGSP, 1920
assert
PMASS, 2900
PMASSFS, 790
ASSOCEQ, 110
associatedEquations, 110
associatedSystem, 110
axServer, 115
getDatabase, 115
uncouplingMatrices, 110
AssociatedEquations, 110
associatedEquations
ASSOCEQ, 110
associatedSystem
ASSOCEQ, 110
atan
EF, 450
STTF, 3449
STTFNC, 3460
atanh
EF, 450
atrapezoidal
NUMQUAD, 2762
att2Result
ESTOOLS, 578
AttachPredicates, 113
axesColorDefault
VIEWDEF, 3720
AxiomServer, 115
AXSERV, 115
multiServ, 115
axServer
ASSOCEQ, 115
B1solve
PLEQN, 2862
badNum
PFOTOOLS, 2981
badValues
PATTERNI, 2892
BalancedFactorisation, 134
balancedFactorisation
BALFACT, 134
BALFACT, 134
balancedFactorisation, 134
bandedHessian
MCALCFN, 1602
bandedJacobian
MCALCFN, 1602
bandMatrix
MAMA, 1543
baseRDE
RDETR, 3592
baseRDEsys
RDETRS, 3597
BasicMethod
INTFACT, 1172
BasicOperatorFunctions1, 136
basis
ALGPKG, 42
INBFF, 1130
basisOfCenter
ALGPKG, 42
basisOfCentroid
ALGPKG, 42
basisOfCommutingElements
ALGPKG, 42
basisOfInterpolateForms
INTFRSP, 1253
basisOfInterpolateFormsForFact
INTFRSP, 1253
basisOfLeftAnnihilator
ALGPKG, 42
basisOfLeftNucleus
ALGPKG, 42
basisOfLeftNucleoid
ALGPKG, 42
basisOfMiddleNucleus
ALGPKG, 42
basisOfNucleus
ALGPKG, 42
basisOfRightAnnihilator
ALGPKG, 42
basisOfRightNucleus
ALGPKG, 42
basisOfRightNucleoid
ALGPKG, 42
bat
TABLBUMP, 3512
bat1
TABLBUMP, 3512
beauzamyBound
GALFACTU, 848
belong?
AF, 14
COMBF, 198
EF, 450
FSPECF, 775
LF, 1432
bernoulli
INTHEORY, 1197
PNTHEORY, 3037
bernoulliB
NTPOLF, 2737
bessel
INDEX

DFSFUN, 348
FSPEC, 775
besselJ
  DFSFUN, 348
  FSPEC, 775
besselK
  DFSFUN, 348
  FSPEC, 775
besselY
  DFSFUN, 348
  FSPEC, 775
Beta
  DFSFUN, 348
  FSPEC, 775
  RDFDIST, 3201
BEZIER, 157
  cubicBezier, 157
  linearBezier, 157
  quadraticBezier, 157
Bezier, 157
BEZOUT, 159
  bezoutDiscriminant, 159
  bezoutMatrix, 159
  bezoutResultant, 159
  brillhartIrreducible?, 174
  brillhartTrials, 174
  sylvesterMatrix, 159
bezoutDiscriminant
  BEZOUT, 159
BezoutMatrix, 159
bezoutMatrix
  BEZOUT, 159
bezoutResultant
  BEZOUT, 159
binaryFunction
  MKBCFUNC, 1454
binomial
  COMBF, 198
  COMBINAT, 1167
  RIDIST, 3204
bipolar
  COORDSYS, 255
bipolarCylindrical
  COORDSYS, 255
biRank
  ALGPKG, 42
biringToPolyRing
 UBLUPPACK, 164
bitCoef
  INTBIT, 1164
bitLength
  INTBIT, 1164
bitTruth
  INTBIT, 1164
bivariate?
  PSETPK, 3047
bivariatePolynomials
  PSETPK, 3047
bivariateSLPEBR
  PFBR, 3006
blockConcat
  MAMA, 1543
blockSplit
  MAMA, 1543
blowUp
  DTP, 303
BlowUpPackage, 164
blowUpWithExcpDiv
  DTP, 303
BLUPPACK, 164
  applyTransform, 164
  biringToPolyRing, 164
  newtonPolySlope, 164
  polyRingToBlUpRing, 164
  quadTransform, 164
  stepBlowUp, 164
bombieriNorm
  GALFACTU, 848
BOP1, 136
  constantOperator, 136
  constantOpIfCan, 136
  derivative, 136
  evaluate, 136
BoundIntegerRoots, 170
boundOfCauchy
  POLUTIL, 3268
BOUNDZRO, 170
  integerBound, 170
branchIfCan
  QCMPACK, 3156
  SFQCMPP, 3394
bright
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>324</td>
</tr>
<tr>
<td>BRILL</td>
<td>174</td>
</tr>
<tr>
<td>noLinearFactor?</td>
<td>174</td>
</tr>
<tr>
<td>brillhartIrreducible?</td>
<td>174</td>
</tr>
<tr>
<td>BEZOUT</td>
<td>174</td>
</tr>
<tr>
<td>BrillhartTests</td>
<td>174</td>
</tr>
<tr>
<td>brillhartTrials</td>
<td>174</td>
</tr>
<tr>
<td>BEZOUT</td>
<td>174</td>
</tr>
<tr>
<td>bringDown</td>
<td>809</td>
</tr>
<tr>
<td>bsolve</td>
<td>2862</td>
</tr>
<tr>
<td>btwFact</td>
<td>853</td>
</tr>
<tr>
<td>BumInSepFFE</td>
<td>853</td>
</tr>
<tr>
<td>UPSQFREE</td>
<td>3678</td>
</tr>
<tr>
<td>bumprow</td>
<td>3512</td>
</tr>
<tr>
<td>TABLBUMP</td>
<td>3512</td>
</tr>
<tr>
<td>bumprow</td>
<td>3512</td>
</tr>
<tr>
<td>bumprow</td>
<td>3512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Library</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>c02aff</td>
<td>NAGC02</td>
<td>2451</td>
</tr>
<tr>
<td>c02agf</td>
<td>NAGC02</td>
<td>2451</td>
</tr>
<tr>
<td>c05adf</td>
<td>NAGC05</td>
<td>2470</td>
</tr>
<tr>
<td>c05nbf</td>
<td>NAGC05</td>
<td>2470</td>
</tr>
<tr>
<td>c05pbf</td>
<td>NAGC05</td>
<td>2470</td>
</tr>
<tr>
<td>c06eaf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06ebf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06ecf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06ekf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06epf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06efq</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>c06erf</td>
<td>NAGC06</td>
<td>2518</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
<tr>
<td>NAGC06</td>
<td>2518</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLOTTOOL</td>
<td>2958</td>
</tr>
<tr>
<td>CYCLES</td>
<td>281</td>
</tr>
<tr>
<td>CARTEN2</td>
<td>178</td>
</tr>
<tr>
<td>C0RDYS</td>
<td>255</td>
</tr>
<tr>
<td>CARTEN2</td>
<td>178</td>
</tr>
<tr>
<td>CDEN</td>
<td>211</td>
</tr>
<tr>
<td>clearDenominator</td>
<td>211</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>324</td>
</tr>
<tr>
<td>central?</td>
<td>427</td>
</tr>
<tr>
<td>D03AGNT</td>
<td>427</td>
</tr>
<tr>
<td>certainlySubVariety?</td>
<td>3047</td>
</tr>
<tr>
<td>PSETPK</td>
<td>3047</td>
</tr>
<tr>
<td>WEIER</td>
<td>3732</td>
</tr>
<tr>
<td>chainSubResultants</td>
<td>3117</td>
</tr>
<tr>
<td>PRS</td>
<td>3117</td>
</tr>
<tr>
<td>PSEUDLIN</td>
<td>3112</td>
</tr>
<tr>
<td>changeName</td>
<td>406</td>
</tr>
<tr>
<td>D01AGNT</td>
<td>406</td>
</tr>
<tr>
<td>changeNameToObjf</td>
<td>629</td>
</tr>
<tr>
<td>E04AGNT</td>
<td>629</td>
</tr>
<tr>
<td>ChangeOfVariable</td>
<td>180</td>
</tr>
<tr>
<td>ODEPRRIC</td>
<td>3103</td>
</tr>
<tr>
<td>characteristicPolynomial</td>
<td>186</td>
</tr>
</tbody>
</table>
CPIMA, 184
EP, 432
NCEP, 2774
NREP, 2780
CharacteristicPolynomialInMonogenicalAlgebra, 184
CharacteristicPolynomialPackage, 186
CHARPOL, 186
characteristicPolynomial, 186
chebyshevT
ORTHPOL, 2813
PNTHEORY, 3037
chebyshevU
ORTHPOL, 2813
PNTHEORY, 3037
check
MULTSQFR, 1615
checkForZero
DFINTTLS, 294
checkPrecision
NAGSP, 1920
checkRur
IRURPK, 1248
chineseRemainder
CRAPACK, 260
IBACHIN, 188
INTHEORY, 1197
ChineseRemainderToolsForIntegralBases, 188
chiSquare
RFDIST, 3201
chiSquare1
RFDIST, 3201
choosemon
LGROBP, 1420
CHVAR, 180
chvar, 180
eval, 180
goodPoint, 180
mkIntegral, 180
radPoly, 180
rootPoly, 180
chvar
CHVAR, 180
Ci
LF, 1432
CINTSLPE, 226
solveLinearPolynomialEquation, 226
classNumber
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
clearCache
SCACHE, 3384
clearDenominator
CDEN, 211
ICDEN, 1102
MCDEN, 1517
UPCDEN, 3664
clearFortranOutputStack
FOP, 747
clikeUniv
WEIER, 3732
CLIP, 3635
clip, 3635
clipBoolean
DROPT0, 399
clipParametric
CLIP, 3635
clipPointsDefault
GRDEF, 948
clipWithRanges
CLIP, 3635
CMPLXRT, 246
cmplxZeros, 246
cn
ELFUTS, 524
coefChoose
MULTSQFR, 1615
coerce
ANY1, 93
CVMP, 193
DRAWHACK, 397
FFHOM, 686
FORMULA1, 3368
MAPPKG1, 1484
NONE1, 2701
INDEX

PAN2EXPR, 2993
PICOERCE, 2954
RESLATC, 3355
RF, 3212
STTAYLOR, 3437
TEX1, 3529
VIEW, 3726
c coerced
CVMP, 193
CoerceVectorMatrixPackage, 193
coleman
SGCF, 3498
color
PTPACK, 2968
columns
MAMA, 1543
COMBF, 198
????, 198
belong?, 198
binomial, 198
commonDenominator, 211
factorial, 198
factorials, 198
iibinom, 198
iidprod, 198
iidsum, 198
ifact, 198
iiperm, 198
iipow, 198
ipow, 198
operator, 198
permutation, 198
product, 198
splitDenominator, 211
summation, 198
COMBINAT, 1167
binomial, 1167
factorial, 1167
multinomial, 1167
partition, 1167
permutation, 1167
stirling1, 1167
stirling2, 1167
CombinatorialFunction, 198
combineFeatureCompatibility
D02AGNT, 420
commaSeparate
D01AGNT, 406
CommonDenominator, 211
commonDenominator
COMBF, 211
ICDEN, 1102
MCDEN, 1517
UPCDEN, 3664
COMMONOP, 213
operator, 213
CommonOperators, 213
COMMUPC, 219
swap, 219
CommuteUnivariatePolynomialCategory, 219
comp
MAPHACK3, 1474
companionBlocks
PSEUDLIN, 3112
compBound
GENEEZ, 924
compdegd
MULTSQFR, 1615
COMPFACC, 221
factor, 221
compiledFunction
MKBCFUNC, 1454
MKUCFUNC, 1468
complete
CYCLES, 281
completeEchelonBasis
REP2, 3338
completeEval
FACUTIL, 647
completeHensel
GHENSEL, 882
completeHermite
SMITH, 3378
completeSmith
SMITH, 3378
COMPLEX2, 224
map, 224
complexEigenvalues
NCEP, 2774
complexEigenvectors
NCEP, 2774
complexElementary
CTRIGMNP, 248
TRIGMNP, 3627
complexExpand
IR2F, 1236
IRRF2F, 1233
ComplexFactorization, 221
complexForm
CTRIGMNP, 248
TRIGMNP, 3627
ComplexFunctions2, 224
ComplexIntegerSolveLinearPolynomialEquation, 226
complexIntegrate
FSCINT, 796
IRRF2F, 1233
complexLimit
LIMITPS, 3077
LIMITRF, 3225
complexNormalize
CTRIGMNP, 248
TRIGMNP, 3627
complexNumeric
NUMERIC, 2740
complexNumericIfCan
NUMERIC, 2740
ComplexPattern, 228
ComplexPatternMatch, 230
ComplexRootFindingPackage, 233
ComplexRootPackage, 246
complexRoots
FLOATCP, 735
complexSolve
FLOATCP, 735
ComplexTrigonometricManipulations, 248
complexZeros
CMPLXRT, 246
CRFP, 233
COMPLPAT, 228
convert, 228
compose
PCOMP, 3001
STTAYLOR, 3437
computeBasis
LGROBP, 1420
computeCycleEntry
CSTTOOLS, 287
computeCycleLength
CSTTOOLS, 287
computeInt
DFINTTLS, 294
concat
ESTOOLS, 578
STREAM1, 3427
conical
COORDSYS, 255
conjugate
PARTPERM, 2888
conjugates
PARTPERM, 2888
consnewpol
MULTSQFR, 1615
const
MAPPKG2, 1495
constant
MAPPKG2, 1495
PFORP, 2838
PMASS, 2900
PMASSFS, 790
constantCoefficientRicDE
ODEPRRIC, 3103
constantIfCan
KERNEL2, 1282
constantKernel
KERNEL2, 1282
constantLeft
MAPPKG3, 1505
ConstantLODE, 252
constantOperator
BOP1, 136
constantOpIfCan
BOP1, 136
constantRight
MAPPKG3, 1505
constantToUnaryFunction
EXPRTUBE, 621
constDsolve
ODECONST, 252
content
HEUGCD, 1067
continuedFraction
NCNTFRAC, 2777
contract
INDEX

IDECOMP, 1074
contractSolve
    SOLVERAD, 3189
convert
    COMPLPAT, 228
    ESTOOLS, 578
    RSCDMPK, 3310
    SRDCMPK, 3404
    ZDSOLVE, 3812
coord
    DROPT0, 399
    LGROBP, 1420
coordinates
    SCPKG, 3466
CoordinateSystems, 255
COORDSYS, 255
    bipolar, 255
    bipolarCylindrical, 255
    cartesian, 255
    conical, 255
    cylindrical, 255
    elliptic, 255
    ellipticCylindrical, 255
    oblateSpheroidal, 255
    parabolic, 255
    parabolicCylindrical, 255
    paraboloidal, 255
    polar, 255
    prolateSpheroidal, 255
    spherical, 255
    toroidal, 255
copies
    DISPLAY, 324
copyfirst
    POLYVEC, 3704
copsylice
    POLYVEC, 3704
cos
    EF, 450
    STTF, 3449
    STTTFNC, 3460
cos2sec
    TRMANIP, 3578
cosh
    EF, 450
cosh2sech
    TRMANIP, 3578
cosSinInfo
    TUBETOOL, 3631
cot
    EF, 450
    STTF, 3449
    STTTFNC, 3460
cot2tan
    TRMANIP, 3578
cot2trig
    TRMANIP, 3578
coth
    EF, 450
coth2tanh
    TRMANIP, 3578
coth2trigh
    TRMANIP, 3578
countRealRoots
    SHP, 3470
countRealRootsMultiple
    SHP, 3470
CPIMA, 184
    characteristicPolynomial, 184
CPMATCH, 230
    patternMatch, 230
CRAPACK, 260
    chineseRemainder, 260
    modTree, 260
    multiEuclideanTree, 260
CRApackage, 260
createGenericMatrix
    REP1, 3330
createIrreduciblePoly
    FFPOLY, 695
createLowComplexityNormalBasis
    FFF, 680
createLowComplexityTable
    FFF, 680
createMultiplicationMatrix
    FFF, 680
createMultiplicationTable
    FFF, 680
createNormalPoly
    FFPOLY, 695
createNormalPrimitivePoly
    FFPOLY, 695
createPrimitiveNormalPoly
    FFPOLY, 695
createPrimitivePoly
    FFPOLY, 695
createRandomElement
    REP2, 3338
createThreeSpace
    TOPSP, 3562
createZechTable
    FFF, 680
credPol
    GBINTERN, 966
crest
    WEIER, 3732
CRFP, 233
    complexZeros, 233
divisorCascade, 233
factor, 233
graeffe, 233
norm, 233
pleskenSplit, 233
reciprocalPolynomial, 233
rootRadius, 233
schwerpunkt, 233
setErrorBound, 233
startPolynomial, 233
critB
    GBINTERN, 966
critBonD
    GBINTERN, 966
critM
    GBINTERN, 966
critMonD1
    GBINTERN, 966
critMTonD1
    GBINTERN, 966
critTopOrder
    GBINTERN, 966
critT
    GBINTERN, 966
cross
    TUBETOOL, 3631
crushedSet
    PSETPK, 3047
csc
    EF, 450

STTF, 3449
STTFNC, 3460
csc2sin
    TRMANIP, 3578
csch
    EF, 450
csch2sinh
    TRMANIP, 3578
CSTTOOLS, 287
    computeCycleEntry, 287
    computeCycleLength, 287
    cycleElt, 287
CTRIGMNP, 248
    complexElementary, 248
    complexForm, 248
    complexNormalize, 248
    imag, 248
    real, 248
    trigs, 248
cubic
    SOLVEFOR, 3065
cubicBezier
    BEZIER, 157
cup
    CYCLES, 281
curry
    MAPPKG2, 1495
curryLeft
    MAPPKG3, 1505
curryRight
    MAPPKG3, 1505
curveColorPalette
    DROPT0, 399
CVMP, 193
    coerce, 193
    coerceP, 193
cycleElt
    CSTTOOLS, 287
CycleIndicators, 281
CYCLES, 281
    alternating, 281
cap, 281
    complete, 281
cup, 281
cyclic, 281
dihedral, 281
INDEX

elementary, 281
eval, 281
graphs, 281
powerSum, 281
SFunction, 281
skewSFunction, 281
wreath, 281
cyclic
CYCLES, 281
cyclicGroup
PGE, 2945
CyclicStreamTools, 287
cyclicSubmodule
REP2, 3338
CYCLOTOM, 290
cyclotomic, 290
cyclotomicDecomposition, 290
cyclotomicFactorization, 290
cyclotomic
CYCLOTOM, 290
NTPOLFN, 2737
PNTHEORY, 3037
cyclotomicDecomposition
CYCLOTOM, 290
cyclotomicFactorization
CYCLOTOM, 290
CyclotomicPolynomialPackage, 290
cylindrical
COORDSYS, 255
d01AgentsPackage, 406
D01AGNT, 406
changeName, 406
commaSeparate, 406
df2st, 406
functionIsContinuousAtEndPoints, 406
functionIsOscillatory, 406
gethi, 406
getlo, 406
ldf2lst, 406
problemPoints, 406
rangeIsFinite, 406
sdf2lst, 406
singularitiesOf, 406
d01ajf
NAGD01, 1999
d01akf
NAGD01, 1999
d01alf
NAGD01, 1999
d01amf
NAGD01, 1999
d01anf
NAGD01, 1999
d01apf
NAGD01, 1999
d01aqf
NAGD01, 1999
d01asf
NAGD01, 1999
d01bbf
NAGD01, 1999
d01fcf
NAGD01, 1999
d01gaf
NAGD01, 1999
d01gbf
NAGD01, 1999
d01WeightsPackage, 413
D01WGTS, 413
exprHasAlgebraicWeight, 413
exprHasLogarithmicWeights, 413
exprHasWeightCosWXorSinWX, 413
d02AgentsPackage, 420
D02AGNT, 420
accuracyIF, 420
combineFeatureCompatibility, 420
eval, 420
expenseOfEvaluationIF, 420
intermediateResultsIF, 420
jacobian, 420
sparsityIF, 420
stiffnessAndStabilityFactor, 420
stiffnessAndStabilityOfODEIF, 420
systemSizeIF, 420
d02bbf
NAGD02, 2386
d02bhf
NAGD02, 2386
d02cjf
NAGD02, 2386
d02cjf
NAGD02, 2386
NAGD02, 2386
d02raf
NAGD02, 2386
d03AgentsPackage, 427
D03AGNT, 427
central?, 427
elliptic?, 427
subscriptedVariables, 427
varList, 427
d03edf
NAGD03, 2433
d03ef
NAGD03, 2433
d03raf
NAGD03, 2433
dAnecdExp
INBFF, 1130
DBLRESP, 390
doubleResultant, 390
DDFACT, 327
distdfact, 328
exptMod, 328
factor, 328
factorSquareFree, 328
irreducible?, 328
separateDegrees, 328
separateFactors, 328
trace2PowMod, 328
tracePowMod, 328
ddFact
MDDFACT, 1556
decompose
MONOTOOL, 1570
PDECOMP, 3003
RSDCMPK, 3310
SRDCMPK, 3404
decomposeFunc
SOLVESER, 3620
definingPolynomial
AF, 14
DefiniteIntegrationTools, 294
DEFINTEF, 468
innerint, 468
integrate, 468
DEFINTRF, 3215
integrate, 3215
degOneCoef
PFORP, 2838
DEGRED, 300
expand, 300
reduce, 300
degree
FACUTIL, 647
PFORP, 2838
POLYVEC, 3704
degreeOfMinimalForm
PFORP, 2838
degreePartition
GALFACT, 853
GALPOLYU, 870
DegreeReductionPackage, 300
degreeSubResultant
PRS, 3117
degreeSubResultantEuclidean
PRS, 3117
denomLODE
ODEPRIM, 3098
denomRicDE
ODEPRRIC, 3103
deriv
STTAYLOR, 3437
derivative
BOP1, 136
desingTree
DTP, 303
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
designtreeAtPoint
DTP, 303
DesingTreePackage, 303
designtreeWoFullParam
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
determinant
INDEX

MATLIN, 1520
df2ef
ESTOOLS, 578
df2fi
ESTOOLS, 578
df2mf
ESTOOLS, 578
df2st
D01AGNT, 406
ESCON'T, 569
ESTOOLS, 578
DFINTTLS, 294
checkForZero, 294
computeInt, 294
ignore?, 294
dflist
ESTOOLS, 578
dfRange
ESTOOLS, 578
DFSFUN, 348
airyAi, 348
airyBi, 348
besselI, 348
besselJ, 348
besselK, 348
besselY, 348
Beta, 348
digamma, 348
E1, 348
Ei, 348
Ei1, 348
Ei2, 348
Ei3, 348
Ei4, 348
Ei5, 348
Ei6, 348
En, 348
fresnelC, 348
fresnelS, 348
Gamma, 348
hypergeomtric0F1, 348
logGamma, 348
polygamma, 348
diag
MAPPKG2, 1495
diagonalMatrix
MAMA, 1543
diagonalProduct
IBATOOL, 1223
diff
ODEINT, 2790
DiffAction
FFFG, 756
DiffC
FFFG, 756
differentiate
POLYVEC, 3704
diffHP
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
digamma
DFSFUN, 348
FSPECF, 775
dihedral
CYCLES, 281
dihedralGroup
PGE, 2945
dilog
LF, 1432
dimensionOfIrreducibleRepresentation
IRSN, 1267
dimensionsOf
NAGSP, 1920
DiophantineSolutionPackage, 313
diophantineSystem
SMITH, 3378
dioSolve
DIOSP, 314
DIOSP, 313
dioSolve, 314
direction
TOOLSIGN, 3531
DirectProductFunctions2, 318
directSum
LODOOPS, 1403
DIRPROD2, 318
map, 318
reduce, 318
scan, 318
DiscreteLogarithmPackage, 321
discriminant
   NFINTBAS, 2726
   PRS, 3117
discriminantEuclidean
   PRS, 3117
DISPLAY, 324
   bright, 324
   center, 324
   copies, 324
   newLine, 324
   say, 324
   sayLength, 324
DisplayPackage, 324
distdfact
   DDFACT, 328
distFact
   LEADCDET, 1317
   DistinctDegreeFactorize, 327
distinguishedCommonRootsOf
   RFP, 3360
distinguishedRootsOf
   RFP, 3360
divergence
   MCALCFN, 1602
divide
   PRS, 3117
divideIfCan
   UPDIVP, 3670
divisorAtDesingTree
   DTP, 303
divisorCascade
   CRFP, 233
divisors
   INTHEORY, 1197
DLP, 321
   shanksDiscLogAlgorithm, 321
dmp2rfi
   PLEQN, 2862
dmpToHdmp
   POLTOPOL, 2987
dmpToP
   POLTOPOL, 2987
dn
   ELFUTS, 524
dot
   TUBETOOL, 3631
double
   REPDB, 3325
doubleDisc
   PFOTOOLS, 2981
DoubleFloatSpecialFunctions, 348
doubleRank
   ALGPKG, 42
doubleResultant
   DBLRESP, 390
DoubleResultantPackage, 390
doublyTransitive?
   ALGFAC?, 52
DRAW, 3533
draw, 3533
   makeObject, 3533
draw
   DRAW, 3533
   DRAWCFUN, 3545
   DRAWCURV, 3541
   DRAWPT, 3559
DRAWCFUN, 3545
draw, 3545
   makeObject, 3545
   recolor, 3545
DrawComplex, 392
drawComplex
   DRAWCX, 392
drawComplexVectorField
   DRAWCX, 392
DRAWCURV, 3541
draw, 3541
drawCurves
   VIEW, 3726
DRAWCX, 392
drawComplex
   DRAWCX, 392
drawComplexVectorField
   DRAWCX, 392
DRAWCURV, 3541
draw, 3541
drawCurves
   VIEW, 3726
drawComplex
   DRAWCX, 392
drawComplexVectorField
   DRAWCX, 392
setClipValue, 392
setImagSteps, 392
setRealSteps, 392
DRAWHACK, 397
coerce, 397
DrawNumericHack, 397
DrawOptionFunctions0, 399
DrawOptionFunctions1, 404
INDEX

DRAWT, 3559
  draw, 3559
drawToScale
  GRDEF, 948
droot
  AF, 14
DROPT0, 399
  adaptive, 399
  clipBoolean, 399
  coord, 399
curveColorPalette, 399
  pointColorPalette, 399
  ranges, 399
  space, 399
  style, 399
title, 399
toScale, 399
tubePoints, 399
tubeRadius, 399
  units, 399
  var1Steps, 399
  var2Steps, 399
  viewpoint, 399
DROPT1, 404
  option, 404
DTP, 303
  adjunctionDivisor, 303
  blowUp, 303
  blowUpWithExcpDiv, 303
desingTree, 303
desingTreeAtPoint, 303
divisorAtDesingTree, 303
  fullParamInit, 303
genus, 303
genusNeg, 303
genusTree, 303
genusTreeNeg, 303
  inBetweenExcpDiv, 303
  initializeParamOfPlaces, 303
  initParLocLeaves, 303

e01baf
  NAGE01, 2047
  e01bf
  NAGE01, 2047
e01bff
  NAGE01, 2047
NAGE02, 1832

\texttt{e02gaf}

NAGE02, 1832

\texttt{e02zaf}

NAGE02, 1832

\texttt{e04AgentsPackage, 629}

\texttt{E04AGNT, 629}

changeNameToObjf, 629
expenseOfEvaluation, 629
finiteBound, 629
linear?, 629
linearMatrix, 629
linearPart, 629
nonLinearPart, 629
optAttributes, 629
quadratic?, 629
simpleBounds?, 629
sortConstraints, 629
splitLinear, 629
sumOfSquares, 629
variables, 629
varList, 629

\texttt{e04dgf}

NAGE04, 2287

\texttt{e04fdf}

NAGE04, 2287

\texttt{e04gcf}

NAGE04, 2287

\texttt{e04jaf}

NAGE04, 2287

\texttt{e04mbf}

NAGE04, 2287

\texttt{e04naf}

NAGE04, 2287

\texttt{e04ucf}

NAGE04, 2287

\texttt{e04ycf}

NAGE04, 2287

\texttt{E1}

DFSFUN, 348

\texttt{edf2df}

ESTOOLS, 578

\texttt{EF, 449}

acos, 450
acosh, 450
acot, 450
acoth, 450
acsc, 450
acsch, 450
asec, 450
asech, 450
asin, 450
asinh, 450
atan, 450
atanh, 450
belong?, 450
cos, 450
cosh, 450
cot, 450
coth, 450
csc, 450
csch, 450
exp, 450
iicos, 450
iicasinh, 450
iicasinh, 450
iicotan, 450
iicotanh, 450
iicos, 450
iicosh, 450
iicot, 450
iicoth, 450
iicsc, 450
iicsech, 450
iiasin, 450
iiasinh, 450
iicatan, 450
iicotanh, 450
iicos, 450
iicosh, 450
iicot, 450
iicoth, 450
iicsc, 450
iicsech, 450
iicexp, 450
iilog, 450
iisec, 450
iisec, 450
iisinh, 450
iisinh, 450
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iisinh</td>
<td>450</td>
</tr>
<tr>
<td>iisqrt2</td>
<td>450</td>
</tr>
<tr>
<td>iisqrt3</td>
<td>450</td>
</tr>
<tr>
<td>itan</td>
<td>450</td>
</tr>
<tr>
<td>itanh</td>
<td>450</td>
</tr>
<tr>
<td>localReal?</td>
<td>450</td>
</tr>
<tr>
<td>log</td>
<td>450</td>
</tr>
<tr>
<td>operator</td>
<td>450</td>
</tr>
<tr>
<td>pi</td>
<td>450</td>
</tr>
<tr>
<td>sec</td>
<td>450</td>
</tr>
<tr>
<td>sech</td>
<td>450</td>
</tr>
<tr>
<td>sin</td>
<td>450</td>
</tr>
<tr>
<td>sinh</td>
<td>450</td>
</tr>
<tr>
<td>specialTrigs</td>
<td>450</td>
</tr>
<tr>
<td>tan</td>
<td>450</td>
</tr>
<tr>
<td>tanh</td>
<td>450</td>
</tr>
<tr>
<td>ef2edf</td>
<td></td>
</tr>
<tr>
<td>EFSTRUC</td>
<td>493</td>
</tr>
<tr>
<td>normalize</td>
<td>493</td>
</tr>
<tr>
<td>realElementary</td>
<td>493</td>
</tr>
<tr>
<td>rischNormalize</td>
<td>493</td>
</tr>
<tr>
<td>rootNormalize</td>
<td>493</td>
</tr>
<tr>
<td>tanQ</td>
<td>493</td>
</tr>
<tr>
<td>validExponential</td>
<td>493</td>
</tr>
<tr>
<td>Ei</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>LF</td>
<td>1432</td>
</tr>
<tr>
<td>Ei1</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>Ei2</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>Ei3</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>Ei4</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>Ei5</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>Ei6</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>eigenMatrix</td>
<td></td>
</tr>
<tr>
<td>REP</td>
<td>3172</td>
</tr>
<tr>
<td>EigenPackage</td>
<td>432</td>
</tr>
<tr>
<td>eigenvalues</td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>432</td>
</tr>
<tr>
<td>eigenvectors</td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>432</td>
</tr>
<tr>
<td>eisensteinIrreducible?</td>
<td>GALFACT, 853</td>
</tr>
<tr>
<td>element</td>
<td></td>
</tr>
<tr>
<td>MAMA</td>
<td>1543</td>
</tr>
<tr>
<td>elementary</td>
<td></td>
</tr>
<tr>
<td>CYCLES</td>
<td>281</td>
</tr>
<tr>
<td>ElementaryFunction</td>
<td>449</td>
</tr>
<tr>
<td>ElementaryFunctionDefiniteIntegration</td>
<td>468</td>
</tr>
<tr>
<td>ElementaryFunctionLODESolver</td>
<td>474</td>
</tr>
<tr>
<td>ElementaryFunctionODESolver</td>
<td>481</td>
</tr>
<tr>
<td>ElementaryFunctionSign</td>
<td>487</td>
</tr>
<tr>
<td>ElementaryFunctionStructurePackage</td>
<td>493</td>
</tr>
<tr>
<td>ElementaryIntegration</td>
<td>502</td>
</tr>
<tr>
<td>ElementaryRischDE</td>
<td>512</td>
</tr>
<tr>
<td>ElementaryRischDESystem</td>
<td>521</td>
</tr>
<tr>
<td>ELFUTS</td>
<td>524</td>
</tr>
<tr>
<td>cn</td>
<td>524</td>
</tr>
<tr>
<td>dn</td>
<td>524</td>
</tr>
<tr>
<td>sn</td>
<td>524</td>
</tr>
<tr>
<td>sncndn</td>
<td>524</td>
</tr>
<tr>
<td>elliptic</td>
<td></td>
</tr>
<tr>
<td>COORDSYS</td>
<td>255</td>
</tr>
<tr>
<td>elliptic?</td>
<td></td>
</tr>
<tr>
<td>D03AGNT</td>
<td>427</td>
</tr>
<tr>
<td>EllipticFunctionsUnivariateTaylorSeries</td>
<td>524</td>
</tr>
<tr>
<td>En</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>348</td>
</tr>
<tr>
<td>enterInCache</td>
<td></td>
</tr>
<tr>
<td>SCACHE</td>
<td>3384</td>
</tr>
<tr>
<td>EP</td>
<td>432</td>
</tr>
<tr>
<td>characteristicPolynomial</td>
<td>432</td>
</tr>
<tr>
<td>eigenvalues</td>
<td>432</td>
</tr>
<tr>
<td>eigenvector</td>
<td>432</td>
</tr>
<tr>
<td>eigenvectors</td>
<td>432</td>
</tr>
<tr>
<td>generalizedEigenvector</td>
<td>432</td>
</tr>
<tr>
<td>generalizedEigenvectors</td>
<td>432</td>
</tr>
<tr>
<td>EQ2</td>
<td>527</td>
</tr>
<tr>
<td>map</td>
<td>527</td>
</tr>
<tr>
<td>EquationFunctions2</td>
<td>527</td>
</tr>
<tr>
<td>erf</td>
<td></td>
</tr>
<tr>
<td>LF</td>
<td>1432</td>
</tr>
</tbody>
</table>
ERROR, 529
   error, 529
error
   ERROR, 529
ErrorFunctions, 529
ES1, 596
   map, 596
ES2, 598
   map, 598
ESCONT, 569
   df2st, 569
functionIsFracPolynomial?, 569
   gethi, 569
   getlo, 569
   ldf2lst, 569
   polynomialZeros, 569
   problemPoints, 569
   sdf2lst, 569
   singularitiesOf, 569
   zerosOf, 569
ESCONT1, 576
   in?, 576
ESTOOLS, 578
   att2Result, 578
   concat, 578
   convert, 578
   df2ef, 578
   df2fi, 578
   df2mf, 578
   df2st, 578
   dflist, 578
   dfRange, 578
   edf2df, 578
   edf2ef, 578
   edf2fi, 578
   ef2edf, 578
   expenseOfEvaluation, 578
   f2df, 578
   f2st, 578
   fi2df, 578
   gethi, 578
   getlo, 578
   iflist2Result, 578
   in?, 578
   isQuotient, 578
   ldf2lst, 578
   ldf2vmf, 578
   mat, 578
   measure2Result, 578
   numberOfOperations, 578
   ocf2ocdf, 578
   outputMeasure, 578
   pdf2df, 578
   pdf2ef, 578
   sdf2lst, 578
   socf2socdf, 578
   vdf2vef, 578
ESTOOLS1, 587
   neglist, 587
ESTOOLS2, 589
   map, 589
   euclideanGroebner
      GBEUCLID, 555
EuclideanGroebnerBasisPackage, 555
   euclideanNormalForm
      GBEUCLID, 555
euler
   INTHEORY, 1197
   PNTHEORY, 3037
eulerE
      NTPOLFN, 2737
eulerPhi
      INTHEORY, 1197
eval
   CHVAR, 180
   CYCLES, 281
   D02AGNT, 420
   EVALCYC, 567
   GPAFF, 889
   PAFF, 2821
   PAFFFF, 2829
   PLPKCRV, 2984
   RF, 3212
   STTAYLOR, 3437
evalADE
   RECOP, 3289
evalat
   POLYVEC, 3704
EVALCYC, 567
eval, 567
evalIfCan
INDEX

GPAFF, 889
PAFF, 2821
PAFFFF, 2829
evalRec
    RECOP, 3289
evaluate
    BOP1, 136
EvaluateCycleIndicators, 567
evenInfiniteProduct
    INPRODO, 1085
    INPRODFF, 1087
    INPRODPF, 1091
    STINPROD, 3434
evelambert
    STTAYLOR, 3437
exp
    EF, 450
    STTF, 3449
    STTFNC, 3460
    XEXPPKG, 3744
EXP3D, 626
    writeObj, 626
expand
    DEGRED, 300
    IR2F, 1236
    IRRF2F, 1233
    TRMANIP, 3578
expandLog
    TRMANIP, 3578
expandPower
    TRMANIP, 3578
expandTrigProducts
    TRMANIP, 3578
expenseOfEvaluation
    E04AGNT, 629
    ESTOOLS, 578
expenseOfEvaluationIF
    D02AGNT, 420
ExpertSystemContinuityPackage, 569
ExpertSystemContinuityPackage1, 576
ExpertSystemToolsPackage, 578
ExpertSystemToolsPackage1, 587
ExpertSystemToolsPackage2, 589
exptendedint
    INTTR, 3567
expint
    ODEINT, 2790
    expintegrate
    INTTR, 3567
    expintfddpoly
    INTTR, 3567
    explimitedint
    INTTR, 3567
    explogs2trigs
    ITRIGMNP, 1154
    exponential
    RFDIST, 3201
    exponential1
    RFDIST, 3201
Export3D, 626
expPot
    INBFF, 1130
EXPR2, 591
    map, 591
EXPR2UPS, 613
    laurent, 613
    puiseux, 613
    series, 613
taylor, 613
ExpressionFunctions2, 591
ExpressionSolve, 593
ExpressionSpaceFunctions1, 596
ExpressionSpaceFunctions2, 598
ExpressionSpaceODESolver, 600
ExpressionToOpenMath, 606
ExpressionToUnivariatePowerSeries, 613
ExpressionTubePlot, 621
exprHasAlgebraicWeight
    D01WGTS, 413
exprHasLogarithmicWeights
    D01WGTS, 413
exprHasWeightCosWXorSinWX
    D01WGTS, 413
EXPRODE, 600
    seriesSolve, 600
EXPRSOL, 593
    replaceDiffs, 593
    seriesSolve, 593
exprToGenUPS
    FS2UPS, 826
exprToUPS
    FS2UPS, 826
INDEX

exprToXXP
FS2EXPXP, 814

EXPRTUBE, 621
constantToUnaryFunction, 621
tubePlot, 621

expt
REPSQ, 3328

exptMod
DDFACT, 328
MDDFACT, 1556

exquo
PRS, 3117
STTAYLOR, 3437

extendededged
POLYVEC, 3704
extendedint
INTRAT, 3241
extendedIntegrate
INTRF, 3223
extractIfCan
TBCMPPK, 3516

f
RFDIST, 3201
f01brf
NAGF01, 2128
f01bsf
NAGF01, 2128
f01maf
NAGF01, 2128
f01mcf
NAGF01, 2128
f01qcf
NAGF01, 2128
f01qdf
NAGF01, 2128
f01qef
NAGF01, 2128
f01qef
NAGF01, 2128
f01rcl
NAGF01, 2128
f01rdf
NAGF01, 2128
f01ref
NAGF01, 2128
f02abf
NAGF02, 1690
f02adef
NAGF02, 1690
f02aeef
NAGF02, 1690
f02af
NAGF02, 1690
f02agf
NAGF02, 1690
f02ajf
NAGF02, 1690
f02akf
NAGF02, 1690
f02awf
NAGF02, 1690
f02axf
NAGF02, 1690
f02bbf
NAGF02, 1690
f02bjf
NAGF02, 1690
f02bff
NAGF02, 1690
f02wef
NAGF02, 1690
f02wef
NAGF02, 1690
f02xsf
NAGF02, 1690
f04adef
NAGF04, 1911
f04arf
NAGF04, 1911
f04asf
NAGF04, 1911
f04atf
NAGF04, 1911
f04bpf
NAGF04, 1911
f04bpf
NAGF04, 1911
f04ref
NAGF04, 1911
f04mef
NAGF04, 1911
f04mef
NAGF04, 1911
f04maf
NAGF04, 1911
INDEX

f04mcf
  NAGF04, 1911
f04qaf
  NAGF04, 1911
f07adf
  NAGF07, 2068
f07aef
  NAGF07, 2068
f07fdf
  NAGF07, 2068
f07fef
  NAGF07, 2068
f2df
  ESTOOLS, 578
F2EXPR
  GUESSF1, 1051
F2FG
  ITRIGMNP, 1154
f2st
  ESTOOLS, 578
FACTEXT, 650
  factor, 650
  factorSqFree, 650
FACTFUNC, 638
  log, 638
  nthRoot, 638
factor
  ALGFACT, 52
  ALGMFACT, 39
  COMPFACT, 221
  CRFP, 233
  DDFACT, 328
  FACTEXT, 650
  FACTRN, 654
  FFFACTOR, 668
  FFFACTSE, 674
  GALFACT, 853
  GAUSSFAC, 877
  GENMFACT, 886
  GENUFACT, 929
  IALGFACT, 1099
  INNMFACT, 1120
  INTFACT, 1172
  LODOF, 1399
  MDDFACT, 1556
  MFINFAC, 1589
  MPCPF, 1575
  MPRFF, 1578
  MRATFAC, 1586
  MULTFACT, 1607
  RATFACT, 3209
  RFFACT, 3218
  RFFACTOR, 3221
  SAEFAC, 3374
  SAERFFC, 3366
  SUPFRACF, 3483
  UNIFACT, 3650
factor1
  LODOF, 1399
factorByRecursion
  PFBR, 3006
  PFBRU, 3013
factorCantorZassenhaus
  FFFACTOR, 668
  FFFACTSE, 674
  FactoredFunctions, 638
  FactoredFunctions2, 642
  FactoredFunctionUtilities, 644
  factorGroebnerBasis
    GBF, 957
factorial
  COMBF, 198
  COMBINAT, 1167
factorials
  COMBF, 198
  FactoringUtilities, 646
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
  650
FactoredFunctionsOverPseudoAlgebraicClosureOfRationalNumber, 654
factorList
  IBACHIN, 188
factorOfDegree
  GALFACT, 853
  GALPOLYU, 870
factorset
  PLEQN, 2862
factorSFBRlcUnit
  PFBR, 3006
  PFBRU, 3013
factorsOfDegree
  GALPOLYU, 870
factorSqFree
   FACTEXT, 650
   FACTRN, 654
factorSquareFree
   DDFACT, 328
   FFFACTOR, 668
   FFFACTSE, 674
   GALFACT, 853
   RATFACT, 3209
   UNIFACT, 3650
factorSquareFreeByRecursion
   PFBR, 3006
   PFBRU, 3013
factorUsingMusser
   FFFACTOR, 668
   FFFACTSE, 674
factorUsingYun
   FFFACTOR, 668
   FFFACTSE, 674
FACTRN, 654
   factor, 654
   factorSqFree, 654
FACUTIL, 646
   completeEval, 647
   degree, 647
   lowerPolynomial, 647
   normalDeriv, 647
   raisePolynomial, 647
   ran, 647
   variables, 647
FAMR2, 664
   map, 664
FCPAK1, 743
   identitySquareMatrix, 743
   zeroMatrix, 743
   zeroSquareMatrix, 743
   zeroVector, 743
FDIV2, 666
   map, 666
ffactor
   FSUPFACT, 843
FFCAT2, 784
   map, 784
FFF, 680
   createLowComplexityNormalBasis, 680
   createLowComplexityTable, 680
createMultiplicationMatrix, 680
createMultiplicationTable, 680
createZechTable, 680
sizeMultiplication, 680
FFFACTOR, 668
   factor, 668
   factorCantorZassenhaus, 668
   factorSquareFree, 668
   factorUsingMusser, 668
   factorUsingYun, 668
   irreducible?, 668
FFFACTSE, 674
   factor, 674
   factorCantorZassenhaus, 674
   factorSquareFree, 674
   factorUsingMusser, 674
   factorUsingYun, 674
   irreducible?, 674
FFFG, 756
   DiffAction, 756
   DiffC, 756
   fffg, 756
   generalCoefficient, 756
   generalInterpolation, 756
   interpolate, 756
   qShiftAction, 756
   qShiftC, 756
   ShiftAction, 756
   ShiftC, 756
ffig
   FFFG, 756
FFFGF, 767
   generalInterpolation, 767
FFHOM, 686
   coerce, 686
FFINTBAS, 786
   integralBasis, 787
   localIntegralBasis, 787
FFPOLY, 695
   createIrreduciblePoly, 695
   createNormalPoly, 695
   createNormalPrimitivePoly, 695
   createPrimitiveNormalPoly, 695
   createPrimitivePoly, 695
   leastAffineMultiple, 695
   nextIrreduciblePoly, 695
nextNormalPoly, 695
nextNormalPrimitivePoly, 695
nextPrimitiveNormalPoly, 695
nextPrimitivePoly, 695
normal?, 695
numberOfIrreduciblePoly, 695
numberOfNormalPoly, 695
numberOfPrimitivePoly, 695
primitive?, 695
random, 695
reducedQPowers, 695

FFPOLY2, 716

rootOfIrreduciblePoly, 716

FFSLEPE, 720

solveLinearIrreduciblePoly, 720

FFSQFR, 722

PolK, 722

Yun, 722

FG2F

ITRIGMNP, 1154

FGLMICPK, 658

fglmIfCan, 658
groebner, 658
zeroDimensional?, 658

fglmIfCan

FGLMICPK, 658
LEXTRIPK, 1390

FGLMICPKPackage, 658

f2df

ESTOOLS, 578

fibonacci

INTHEORY, 1197

fillPascalTriangle

GALUTIL, 873

FindOrderFinite, 662

findOrderOfDivisor

GPAFF, 889
PAFF, 2821
PAFFFF, 2829

FiniteAbelianMonoidRingFunctions2, 664

finiteBound

E04AGNT, 629

FiniteDivisorFunctions2, 666

FiniteFieldFactorization, 668

FiniteFieldFactorizationWithSizeParseBySideEffect, 674

FiniteFieldFunctions, 680

FiniteFieldHomomorphisms, 686

FiniteFieldPolynomialPackage, 695

FiniteFieldPolynomialPackage2, 716

FiniteFieldSolveLinearPolynomialEquation, 720

FiniteFieldSquareFreeDecomposition, 722

FiniteLinearAggregateFunctions2, 726

FiniteLinearAggregateSort, 729

finiteSeries2LinSys

LISYSE, 1409

finiteSeries2LinSysWQVectorise

LISYSE, 1409

finiteSeries2Vector

LISYSE, 1409

FiniteSetAggregateFunctions2, 733

firstExponent

PQORP, 2838

firstSubsetGray

GRAY, 951

firstUncouplingMatrix

PREASSOC, 3089

fixedDivisor

PNTHEORY, 3037

fixedPoint

MAPPKG1, 1484

fixedPointExquo

UTSODE, 3690

fixPredicates

PMDOWN, 2925

FLAGG2, 726

map, 726

reduce, 726

scan, 726

FLASORT, 729

heapSort, 729

quickSort, 729

shellSort, 729

FLOATCP, 735

complexRoots, 735

cpolve, 735

FloatingComplexPackage, 735

FloatingRealPackage, 739

FLOATRP, 739

realRoots, 739

FOP, 747
clearFortranOutputStack, 747
popFortranOutputStack, 747
pushFortranOutputStack, 747
showFortranOutputStack, 747
topFortranOutputStack, 747
FORDER, 662
order, 662
FormatArabic
NUMFMT, 2732
FormatRoman
NUMFMT, 2732
FORMULA1, 3368
coerce, 3368
FORT, 750
linkToFortran, 750
outputAsFortran, 750
setLegalFortranSourceExtensions, 750
FortranCodePackage1, 743
fortranCompilerName
NAGSP, 1920
fortranLinkerArgs
NAGSP, 1920
FortranOutputStackPackage, 747
FortranPackage, 750
foundZeros
RFP, 3360
fpfindINFO
GBINTERN, 966
FR2, 642
map, 642
FRAC2, 771
map, 771
FractionalIdealFunctions2, 753
FractionFreeFastGaussian, 756
FractionFreeFastGaussianFractions, 767
FractionFunctions2, 771
FramedNonAssociativeAlgebraFunctions2, 773
fresnelC
        DFSFUN, 348
fresnelS
        DFSFUN, 348
FRIDEAL2, 753
map, 753
FRNAAF2, 773
map, 773
Frobenius
        NORMRETR, 2719
        froot
        POLYROOT, 3042
        FRUTIL, 644
        mergeFactors, 644
        refine, 644
        FS2, 799
        map, 799
        FS2EXPXP, 814
        exprToXXP, 814
        localAbs, 814
        FS2UPS, 826
        exprToGenUPS, 826
        exprToUPS, 826
        localAbs, 826
        FSAGG2, 733
        map, 733
        reduce, 733
        scan, 733
        FSCINT, 796
        complexIntegrate, 796
        internalIntegrate, 796
        internalIntegrate0, 796
        FSINT, 801
        integrate, 801
        FSPECF, 775
        abs, 775
        airyAi, 775
        airyBi, 775
        belong?, 775
        besselI, 775
        besselJ, 775
        besselK, 775
        besselY, 775
        Beta, 775
digamma, 775
Gamma, 775
iiabs, 775
iiairyAi, 775
iiairyBi, 775
iibesselI, 775
iibesselJ, 775
iibesselK, 775
iibesselY, 775
iibeta, 775
iidigamma, 775
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iiGamma</td>
<td>775</td>
</tr>
<tr>
<td>iipolygamma</td>
<td>775</td>
</tr>
<tr>
<td>operator</td>
<td>775</td>
</tr>
<tr>
<td>polygamma</td>
<td>775</td>
</tr>
<tr>
<td>FSPRMELT</td>
<td>805</td>
</tr>
<tr>
<td>primitiveElement</td>
<td>805</td>
</tr>
<tr>
<td>FSRED</td>
<td>809</td>
</tr>
<tr>
<td>bringDown</td>
<td>809</td>
</tr>
<tr>
<td>newReduc</td>
<td>809</td>
</tr>
<tr>
<td>FSUPFACT</td>
<td>843</td>
</tr>
<tr>
<td>anfactor</td>
<td>843</td>
</tr>
<tr>
<td>ffactor</td>
<td>843</td>
</tr>
<tr>
<td>qfactor</td>
<td>843</td>
</tr>
<tr>
<td>UP2ifCan</td>
<td>843</td>
</tr>
<tr>
<td>fullDesTree</td>
<td></td>
</tr>
<tr>
<td>PAFF</td>
<td>2821</td>
</tr>
<tr>
<td>PAFFFF</td>
<td>2829</td>
</tr>
<tr>
<td>fullInfClsPt</td>
<td></td>
</tr>
<tr>
<td>PAFF</td>
<td>2821</td>
</tr>
<tr>
<td>PAFFFF</td>
<td>2829</td>
</tr>
<tr>
<td>fullParamInit</td>
<td></td>
</tr>
<tr>
<td>DTP</td>
<td>303</td>
</tr>
<tr>
<td>function</td>
<td></td>
</tr>
<tr>
<td>MKFUNC</td>
<td>1464</td>
</tr>
<tr>
<td>FunctionalSpecialFunction</td>
<td>775</td>
</tr>
<tr>
<td>FunctionFieldCategoryFunctions2</td>
<td>784</td>
</tr>
<tr>
<td>FunctionFieldIntegralBasis</td>
<td>786</td>
</tr>
<tr>
<td>functionIsContinuousAtEndPoints</td>
<td></td>
</tr>
<tr>
<td>D01AGNT</td>
<td>406</td>
</tr>
<tr>
<td>functionIsFracPolynomial?</td>
<td></td>
</tr>
<tr>
<td>ESCONT</td>
<td>569</td>
</tr>
<tr>
<td>functionIsOscillatory</td>
<td></td>
</tr>
<tr>
<td>D01AGNT</td>
<td>406</td>
</tr>
<tr>
<td>FunctionSpaceAssertions</td>
<td>790</td>
</tr>
<tr>
<td>FunctionSpaceAttachPredicates</td>
<td>793</td>
</tr>
<tr>
<td>FunctionSpaceComplexIntegration</td>
<td>796</td>
</tr>
<tr>
<td>FunctionSpaceFunctions2</td>
<td>799</td>
</tr>
<tr>
<td>FunctionSpaceIntegration</td>
<td>801</td>
</tr>
<tr>
<td>FunctionSpacePrimitiveElement</td>
<td>805</td>
</tr>
<tr>
<td>FunctionSpaceReduce</td>
<td>809</td>
</tr>
<tr>
<td>FunctionSpaceSum</td>
<td>811</td>
</tr>
<tr>
<td>FunctionSpaceToExponentialExpansion</td>
<td>814</td>
</tr>
<tr>
<td>FunctionSpaceToUnivariatePowerSeries</td>
<td>826</td>
</tr>
<tr>
<td>FunctionSpaceUnivariatePolynomialFactor</td>
<td>843</td>
</tr>
<tr>
<td>GALFACT</td>
<td>852</td>
</tr>
<tr>
<td>btwFact</td>
<td>853</td>
</tr>
<tr>
<td>degreePartition</td>
<td>853</td>
</tr>
<tr>
<td>eisensteinIrreducible?</td>
<td>853</td>
</tr>
<tr>
<td>factor</td>
<td>853</td>
</tr>
<tr>
<td>factorOfDegree</td>
<td>853</td>
</tr>
<tr>
<td>factorSquareFree</td>
<td>853</td>
</tr>
<tr>
<td>henselFact</td>
<td>853</td>
</tr>
<tr>
<td>makeFR</td>
<td>853</td>
</tr>
<tr>
<td>modularFactor</td>
<td>853</td>
</tr>
<tr>
<td>musserTrials</td>
<td>853</td>
</tr>
<tr>
<td>numberOfFactors</td>
<td>853</td>
</tr>
<tr>
<td>stopMusserTrials</td>
<td>853</td>
</tr>
<tr>
<td>tryFunctionalDecomposition</td>
<td>853</td>
</tr>
<tr>
<td>tryFunctionalDecomposition?</td>
<td>853</td>
</tr>
<tr>
<td>useEisensteinCriterion</td>
<td>853</td>
</tr>
<tr>
<td>useEisensteinCriterion?</td>
<td>853</td>
</tr>
<tr>
<td>useSingleFactorBound</td>
<td>853</td>
</tr>
<tr>
<td>useSingleFactorBound?</td>
<td>853</td>
</tr>
<tr>
<td>GALFACTU</td>
<td>848</td>
</tr>
<tr>
<td>beauzamyBound</td>
<td>848</td>
</tr>
<tr>
<td>bombieriNorm</td>
<td>848</td>
</tr>
<tr>
<td>height</td>
<td>848</td>
</tr>
<tr>
<td>infinityNorm</td>
<td>848</td>
</tr>
<tr>
<td>length</td>
<td>848</td>
</tr>
<tr>
<td>norm</td>
<td>848</td>
</tr>
<tr>
<td>quadraticNorm</td>
<td>848</td>
</tr>
<tr>
<td>rootBound</td>
<td>848</td>
</tr>
<tr>
<td>singleFactorBound</td>
<td>848</td>
</tr>
<tr>
<td>GaloisGroupFactorizationUtilities</td>
<td></td>
</tr>
<tr>
<td>GaloisGroupFactorizer</td>
<td>852</td>
</tr>
<tr>
<td>GaloisGroupPolynomialUtilities</td>
<td>870</td>
</tr>
<tr>
<td>GaloisGroupUtilities</td>
<td>873</td>
</tr>
<tr>
<td>GALPOLYU</td>
<td>870</td>
</tr>
<tr>
<td>degreePartition</td>
<td>870</td>
</tr>
<tr>
<td>factorOfDegree</td>
<td>870</td>
</tr>
<tr>
<td>factorsOfDegree</td>
<td>870</td>
</tr>
<tr>
<td>monic?</td>
<td>870</td>
</tr>
<tr>
<td>reverse</td>
<td>870</td>
</tr>
<tr>
<td>scaleRoots</td>
<td>870</td>
</tr>
<tr>
<td>shiftRoots</td>
<td>870</td>
</tr>
<tr>
<td>unvectorise</td>
<td>870</td>
</tr>
<tr>
<td>GALUTIL</td>
<td>873</td>
</tr>
<tr>
<td>fillPascalTriangle</td>
<td>873</td>
</tr>
<tr>
<td>pascalTriangle</td>
<td>873</td>
</tr>
<tr>
<td>rangePascalTriangle</td>
<td>873</td>
</tr>
<tr>
<td>safeCeiling</td>
<td>873</td>
</tr>
</tbody>
</table>
safeFloor, 873
safetyMargin, 873
sizePascalTriangle, 873
Gamma
DFSFUN, 348
FSPECF, 775
GAUSSFAC, 877
factor, 877
prime?, 877
sumSquares, 877
GaussianFactorizationPackage, 877
GB, 1002
groebner, 1002
normalForm, 1002
gbasis
GBINTERN, 966
GBEUCLID, 555
euclideanGroebner, 555
euclideanNormalForm, 555
GBF, 957
factorGroebnerBasis, 957
groebnerFactorize, 957
GBINTERN, 965
credPol, 966
critB, 966
critBonD, 966
critM, 966
critMonD1, 966
critMTonD1, 966
critpOrder, 966
critT, 966
fprindINFO, 966
gbasis, 966
hMonic, 966
lepol, 966
makeCrit, 966
minGbasis, 966
prinb, 966
prindINFO, 966
prinpollINFO, 966
prinshINFO, 966
redPo, 966
redPol, 966
sPol, 966
updatD, 966
updatF, 966
virtualDegree, 966
gcd
HEUGCD, 1067
MDDFACT, 1556
PGCD, 3022
POLYVEC, 3704
PRS, 3117
gcdcofact
HEUGCD, 1067
gcdcofactprim
HEUGCD, 1067
gcdPolynomial
GENPGCD, 904
gcdprim
HEUGCD, 1067
gcdPrimitive
PGCD, 3022
gderiv
STTAYLOR, 3437
GDRAW, 939
GospersMethod, 939
GENEEZ, 924
compBound, 924
reduction, 924
solveid, 924
tablePow, 924
testModulus, 924
generalCoefficient
FFFG, 756
GeneralHenselPackage, 882
generalInfiniteProduct
INPRODO, 1085
INPRODF, 1087
INPRODPF, 1091
STINPROD, 3434
generalInterpolation
FFFG, 756
FFFGF, 767
generalizedEigenvector
EP, 432
generalizedEigenvectors
EP, 432
GeneralizedMultivariateFactorize, 886
generalLambert
STTAYLOR, 3437
GeneralPackageForAlgebraicFunctionField, 889
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneralPolynomialGcdPackage, genEq</td>
<td>904</td>
</tr>
<tr>
<td>generalSqFr, TWOFACT, 3642</td>
<td></td>
</tr>
<tr>
<td>generalTwoFactor, TWOFACT, 3642</td>
<td></td>
</tr>
<tr>
<td>generate, HB, 1064</td>
<td></td>
</tr>
<tr>
<td>generateIrredPoly, IRREDFFX, 1264</td>
<td></td>
</tr>
<tr>
<td>GenerateUnivariatePowerSeries, 918</td>
<td></td>
</tr>
<tr>
<td>genericPosition, GROEBSOL, 1006</td>
<td></td>
</tr>
<tr>
<td>GenExEuclid, 924</td>
<td></td>
</tr>
<tr>
<td>GENMFACT, 886</td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td>886</td>
</tr>
<tr>
<td>GENPGCD, 904</td>
<td></td>
</tr>
<tr>
<td>gcdPolynomial, 904</td>
<td></td>
</tr>
<tr>
<td>randomR, 904</td>
<td></td>
</tr>
<tr>
<td>GENUFACT, 929</td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td>929</td>
</tr>
<tr>
<td>GenUFactorize, 929</td>
<td></td>
</tr>
<tr>
<td>GENUPS, 918</td>
<td></td>
</tr>
<tr>
<td>laurent, 918</td>
<td></td>
</tr>
<tr>
<td>puiseux, 918</td>
<td></td>
</tr>
<tr>
<td>series, 918</td>
<td></td>
</tr>
<tr>
<td>taylor, 918</td>
<td></td>
</tr>
<tr>
<td>genus, DTP, 303</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td></td>
</tr>
<tr>
<td>PAFF, 2821</td>
<td></td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td></td>
</tr>
<tr>
<td>genusNeg, DTP, 303</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td></td>
</tr>
<tr>
<td>PAFF, 2821</td>
<td></td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td></td>
</tr>
<tr>
<td>genusTree, DTP, 303</td>
<td></td>
</tr>
<tr>
<td>genusTreeNeg, DTP, 303</td>
<td></td>
</tr>
<tr>
<td>GenusZeroIntegration, 932</td>
<td></td>
</tr>
<tr>
<td>geometric, RIDIST, 3204</td>
<td></td>
</tr>
<tr>
<td>getDatabase, ASSOCEQ, 115</td>
<td></td>
</tr>
<tr>
<td>OPQUERY, 2806</td>
<td></td>
</tr>
<tr>
<td>getEq, RECOP, 3289</td>
<td></td>
</tr>
<tr>
<td>getGoodPrime, PFOTOOLS, 2981</td>
<td></td>
</tr>
<tr>
<td>gethi, D01AGNT, 406</td>
<td></td>
</tr>
<tr>
<td>ESCONT, 569</td>
<td></td>
</tr>
<tr>
<td>ESTOOLS, 578</td>
<td></td>
</tr>
<tr>
<td>getlo, D01AGNT, 406</td>
<td></td>
</tr>
<tr>
<td>ESCONT, 569</td>
<td></td>
</tr>
<tr>
<td>ESTOOLS, 578</td>
<td></td>
</tr>
<tr>
<td>getOp, RECOP, 3289</td>
<td></td>
</tr>
<tr>
<td>getOrder, UDPO, 3696</td>
<td></td>
</tr>
<tr>
<td>getShiftRec, RECOP, 3289</td>
<td></td>
</tr>
<tr>
<td>getVariableOrder, UDVO, 3699</td>
<td></td>
</tr>
<tr>
<td>GF2FG, ITRIGMNP, 1154</td>
<td></td>
</tr>
<tr>
<td>GHENSEL, 882</td>
<td></td>
</tr>
<tr>
<td>completeHensel, 882</td>
<td></td>
</tr>
<tr>
<td>HenselLift, 882</td>
<td></td>
</tr>
<tr>
<td>reduction, 882</td>
<td></td>
</tr>
<tr>
<td>GnuDraw, 939</td>
<td></td>
</tr>
<tr>
<td>goodnessOfFit, OPTPACK, 67</td>
<td></td>
</tr>
<tr>
<td>goodPoint, CHVAR, 180</td>
<td></td>
</tr>
<tr>
<td>goppaCode, PAFF, 2821</td>
<td></td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td></td>
</tr>
<tr>
<td>GOSPER, 942</td>
<td></td>
</tr>
<tr>
<td>GospersMethod, GDRAW, 939</td>
<td></td>
</tr>
<tr>
<td>GOSPER, 942</td>
<td></td>
</tr>
<tr>
<td>GosperSummationMethod, 942</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td></td>
</tr>
<tr>
<td>adjunctionDivisor, 889</td>
<td></td>
</tr>
<tr>
<td>classNumber, 889</td>
<td></td>
</tr>
<tr>
<td>desingTree, 889</td>
<td></td>
</tr>
<tr>
<td>desingTreeWoFullParam, 889</td>
<td></td>
</tr>
<tr>
<td>eval, 889</td>
<td></td>
</tr>
</tbody>
</table>
evalIfCan, 889
findOrderOfDivisor, 889
genus, 889
genusNeg, 889
homogenize, 889
interpolateForms, 889
interpolateFormsForFact, 889
intersectionDivisor, 889
lBasis, 889
LPolynomial, 889
numberOfPlacesOfDegree, 889
numberPlacesDegExtDeg, 889
numberRatPlacesExtDeg, 889
parametrize, 889
placesAbove, 889
placesOfDegree, 889
pointDominateBy, 889
printInfo, 889
rationalPlaces, 889
rationalPoints, 889
reset, 889
setCurve, 889
setSingularPoints, 889
singularPoints, 889
theCurve, 889
ZetaFunction, 889
gradient
MCALCFN, 1602
graeffe
CRFP, 233
gramschmidt
REP, 3172
graphCurves
VIEW, 3726
GraphicsDefaults, 948
graphs
CYCLES, 281
GRAY, 951
firstSubsetGray, 951
nextSubsetGray, 951
GrayCode, 951
GRDEF, 948
adaptive, 948
clipPointsDefault, 948
drawToScale, 948
maxPoints, 948
minPoints, 948
screenResolution, 948
groebgen
LGROBP, 1420
groebner
FGLMICPK, 658
GB, 1002
INTERGB, 1161
LEXTRIPK, 1390
GroebnerFactorizationPackage, 957
groebnerFactorize
GBF, 957
GroebnerInternalPackage, 965
GroebnerPackage, 1002
GroebnerSolve, 1006
GROEBSOL, 1006
genericPosition, 1006
groebSolve, 1006
testDim, 1006
groebSolve
GROEBSOL, 1006
GUESS, 1012
diffHP, 1012
guess, 1012
guessADE, 1012
guessAlg, 1012
guessBinRat, 1012
guessExpRat, 1012
guessHolo, 1012
guessHP, 1012
guessPade, 1012
guessPRec, 1012
guessRat, 1012
guessRec, 1012
shiftHP, 1012
Guess, 1012
guess
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
guessADE
GUESS, 1012
GUESSAN, 1048
INDEX

GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

guessAlg

GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

GuessAlgebraicNumber, 1048

GUESSAN, 1048
diffHP, 1048
guess, 1048
guessADE, 1048
guessAlg, 1048
guessBinRat, 1048
guessExpRat, 1048
guessHolo, 1048
guessHP, 1048
guessPade, 1048
guessPRec, 1048
guessRec, 1048
shiftHP, 1048

guessBinRat

GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

guessExpRat

GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

GUESSF, 1050
diffHP, 1050
guess, 1050
guessADE, 1050
guessAlg, 1050
guessBinRat, 1050

guessExpRat, 1050
guessHolo, 1050
guessHP, 1050
guessPade, 1050
guessPRec, 1050
guessRec, 1050
shiftHP, 1050

GUESSF1, 1051

F2EXPRR, 1051

GuessFinite, 1050

GuessFiniteFunctions, 1051

guessHolo

GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

guessHP

GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057

GUESSINT, 1053
diffHP, 1053
guess, 1053
guessADE, 1053
guessAlg, 1053
guessBinRat, 1053
guessExpRat, 1053
guessHolo, 1053
guessHP, 1053
guessPade, 1053
guessPRec, 1053
guessRec, 1053
shiftHP, 1053

GuessInteger, 1053

GUESSP, 1055
diffHP, 1055
guess, 1055
guessADE, 1055
guessAlg, 1055
guessBinRat, 1055
guessExpRat, 1055
guessHolo, 1055
guessHP, 1055
guessPade, 1055
guessPRec, 1055
guessRat, 1055
guessRec, 1055
shiftHP, 1055
guessPade
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
GuessPolynomial, 1055
guessPRec
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
guessRat
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
guessRec
GUESS, 1012
GUESSAN, 1048
GUESSF, 1050
GUESSINT, 1053
GUESSP, 1055
GUESSUP, 1057
GuessUnivariatePolynomial, 1057
GUESSUP, 1057
diffHP, 1057
guess, 1057
guessADE, 1057
guessAlg, 1057
guessBinRat, 1057
guessExpRat, 1057
guessHolo, 1057
guessHP, 1057
guessPade, 1057
guessPRec, 1057
guessRat, 1057
guessRec, 1057
shiftHP, 1057
hadamard
UFPS1, 3658
HallBasis, 1064
harmonic
INTHEORY, 1197
hasSoln
PLEQN, 2862
hasSolution?
LSMP, 1412
LSMP1, 1415
Hausdorff
XEXPPKG, 3744
HB, 1064
generate, 1064
inHallBasis?, 1064
ifunc, 1064
hdmpToDmp
POLTOPOL, 2987
hdmpToP
POLTOPOL, 2987
heapSort
FLASORT, 729
height
GALFACTU, 848
henselFact
GALFACT, 853
UNIFACT, 3650
HenselLift
GHENSEL, 882
hermite
PNTHEORY, 3037
SMITH, 3378
hermiteH
ORTHPOL, 2813
HermiteIntegrate
INTHERAL, 20
INTHERTR, 3564
hessian
INDEX

ICDEN, 1102
clearDenominator, 1102
commonDenominator, 1102
splitDenominator, 1102
id
MAPPKG1, 1484
IdealDecompositionPackage, 1074
idealiser
IBATOOL, 1223
idealiserMatrix
IBATOOL, 1223
IDECOMP, 1074
contract, 1074
primaryDecomp, 1074
prime?, 1074
radical, 1074
zeroDimPrimary?, 1074
zeroDimPrime?, 1074
identitySquareMatrix
FCPAK1, 743
iflist2Result
ESTOOLS, 578
ignore?
DFINTTLS, 294
iiabs
FSPECF, 775
iiacos
EF, 450
iiacosinh
EF, 450
iiacot
EF, 450
iiacoth
EF, 450
iiacsc
EF, 450
iiacsch
EF, 450
iiAiryAi
FSPECF, 775
iiAiryBi
FSPECF, 775
iiasec
EF, 450
iiasinh
EF, 450

ICALCFN, 1602
HEUGCD, 1067
cntent, 1067
gcd, 1067
gcdcofact, 1067
gcdcofactprim, 1067
gcdprim, 1067
lintgcd, 1067
HeuGcd, 1067
hMonic
GBINTERN, 966
homogenize
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
PFORP, 2838
horizConcat
MAMA, 1543
horizSplit
MAMA, 1543
htrigs
TRMANIP, 3578
hue
PTPACK, 2968
hypergeometric0F1
DFSFUN, 348
IALGFACT, 1099
factor, 1099
IBACHIN, 188
chineseRemainder, 188
factorList, 188
listConjugateBases, 188
IBATOOL, 1223
diagonalProduct, 1223
divideIfCan, 1223
idealiser, 1223
idealiserMatrix, 1223
leastPower, 1223
matrixGcd, 1223
moduleSum, 1223
IBPTOOLS, 1227
mapBivariate, 1227
mapMatrixIfCan, 1227
mapUnivariate, 1227
mapUnivariateIfCan, 1227
ICDEN, 1102
clearDenominator, 1102
commonDenominator, 1102
splitDenominator, 1102
id
MAPPKG1, 1484
IdealDecompositionPackage, 1074
idealiser
IBATOOL, 1223
idealiserMatrix
IBATOOL, 1223
IDECOMP, 1074
contract, 1074
primaryDecomp, 1074
prime?, 1074
radical, 1074
zeroDimPrimary?, 1074
zeroDimPrime?, 1074
identitySquareMatrix
FCPAK1, 743
iflist2Result
ESTOOLS, 578
ignore?
DFINTTLS, 294
iiabs
FSPECF, 775
iiacos
EF, 450
iiacosinh
EF, 450
iiacot
EF, 450
iiacoth
EF, 450
iiacsc
EF, 450
iiacsch
EF, 450
iiAiryAi
FSPECF, 775
iiAiryBi
FSPECF, 775
iiasec
EF, 450
iiasinh
EF, 450
iiasin  
   EF, 450  
iiasinh  
   EF, 450  
iiatan  
   EF, 450  
iiatanh  
   EF, 450  
iiBesselI  
   FSPECF, 775  
iiBesselJ  
   FSPECF, 775  
iiBesselK  
   FSPECF, 775  
iiBesselY  
   FSPECF, 775  
iiBeta  
   FSPECF, 775  
iibinom  
   COMBF, 198  
iicos  
   EF, 450  
iicosh  
   EF, 450  
iicot  
   EF, 450  
iicoth  
   EF, 450  
iicsc  
   EF, 450  
iicsch  
   EF, 450  
iidigamma  
   FSPECF, 775  
iidprod  
   COMBF, 198  
iidsum  
   COMBF, 198  
iixexp  
   EF, 450  
iifact  
   COMBF, 198  
iGamma  
   FSPECF, 775  
iilog  
   EF, 450  
iiperm  
   COMBF, 198  
iipolygamma  
   FSPECF, 775  
iipow  
   COMBF, 198  
iisec  
   EF, 450  
iisech  
   EF, 450  
iisin  
   EF, 450  
iisinh  
   EF, 450  
iisqrt2  
   EF, 450  
iisqrt3  
   EF, 450  
iiatan  
   EF, 450  
iiatanh  
   EF, 450  
imag  
   CTRIGMNIP, 248  
   TRIGMNIP, 3627  
images  
   AF, 14  
   AFALGGRO, 4  
   AFALGRES, 9  
   ALGFACT, 52  
   ALGMANIP, 34  
   ALGMFACT, 39  
   ALGPKG, 42  
   ANY1, 93  
   API, 101  
   APPLYORE, 108  
   APRRULE, 104  
   ARRAY12, 2796  
   ASSOCEQ, 110  
   AXSERV, 115  
   BALFACT, 134  
   BEZIER, 157  
   BEZOUT, 159  
   BLUPPACK, 164  
   BOP1, 136  
   BOUNDFZRO, 170
<table>
<thead>
<tr>
<th>INDEX</th>
<th>3869</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRILL, 174</td>
<td>DROPT0, 399</td>
</tr>
<tr>
<td>CARTEN2, 178</td>
<td>DROPT1, 404</td>
</tr>
<tr>
<td>CDEN, 211</td>
<td>DTP, 303</td>
</tr>
<tr>
<td>CHARPOL, 186</td>
<td>E04AGNT, 629</td>
</tr>
<tr>
<td>CHVAR, 180</td>
<td>EF, 449</td>
</tr>
<tr>
<td>CINTSLEPE, 226</td>
<td>EFSTRUC, 493</td>
</tr>
<tr>
<td>CLIP, 3635</td>
<td>ELFUTS, 524</td>
</tr>
<tr>
<td>CMPLXRT, 246</td>
<td>EP, 432</td>
</tr>
<tr>
<td>COMBF, 198</td>
<td>EQ2, 527</td>
</tr>
<tr>
<td>COMBINAT, 1167</td>
<td>ERROR, 529</td>
</tr>
<tr>
<td>COMMONOP, 213</td>
<td>ES1, 596</td>
</tr>
<tr>
<td>COMMUPC, 219</td>
<td>ES2, 598</td>
</tr>
<tr>
<td>COMPFACK, 221</td>
<td>ESCONT, 569</td>
</tr>
<tr>
<td>COMPLEX2, 224</td>
<td>ESCONT1, 576</td>
</tr>
<tr>
<td>COMPLPAT, 228</td>
<td>ESTOOLS, 578</td>
</tr>
<tr>
<td>COORDSYS, 255</td>
<td>ESTOOLS1, 587</td>
</tr>
<tr>
<td>CPIMA, 184</td>
<td>ESTOOLS2, 589</td>
</tr>
<tr>
<td>CPMATCH, 230</td>
<td>EVALCyc, 567</td>
</tr>
<tr>
<td>CRAPACK, 260</td>
<td>EXP3D, 626</td>
</tr>
<tr>
<td>CRFF, 233</td>
<td>EXPR2, 591</td>
</tr>
<tr>
<td>CSTTOOLS, 287</td>
<td>EXPR2UPS, 613</td>
</tr>
<tr>
<td>CTRIGMNP, 248</td>
<td>EXPRODE, 600</td>
</tr>
<tr>
<td>CVMP, 193</td>
<td>EXPRSOl, 593</td>
</tr>
<tr>
<td>CYCLES, 281</td>
<td>EXPRTEXT, 650</td>
</tr>
<tr>
<td>CYCLOTOM, 290</td>
<td>FACTFUNC, 638</td>
</tr>
<tr>
<td>D01AGNT, 406</td>
<td>FACTRN, 654</td>
</tr>
<tr>
<td>D01WGTS, 413</td>
<td>FACUTIL, 646</td>
</tr>
<tr>
<td>D02AGNT, 420</td>
<td>FAMR2, 664</td>
</tr>
<tr>
<td>D03AGNT, 427</td>
<td>FFACCTOR, 668</td>
</tr>
<tr>
<td>DBLRESP, 390</td>
<td>FFFACTSE, 674</td>
</tr>
<tr>
<td>DDFACT, 327</td>
<td>FFFFG, 756</td>
</tr>
<tr>
<td>DEFINTEF, 468</td>
<td>FFFGF, 767</td>
</tr>
<tr>
<td>DEFINTRF, 3215</td>
<td>FFHOM, 686</td>
</tr>
<tr>
<td>DEGRED, 300</td>
<td>FFINTBAS, 787</td>
</tr>
<tr>
<td>DFINTLS, 294</td>
<td>FFPOLy, 695</td>
</tr>
<tr>
<td>DFSFUN, 348</td>
<td>FFPOLy2, 716</td>
</tr>
<tr>
<td>DIOSP, 313</td>
<td>FFSLPE, 720</td>
</tr>
<tr>
<td>DIRPROD2, 318</td>
<td>FFSQFR, 722</td>
</tr>
<tr>
<td>DISPLAY, 324</td>
<td>FGLMICPK, 658</td>
</tr>
<tr>
<td>DLP, 321</td>
<td>FLAGG2, 726</td>
</tr>
<tr>
<td>DRAW, 3533</td>
<td>FLASORT, 729</td>
</tr>
<tr>
<td>DRAWCFUN, 3545</td>
<td>FFPOlY, 695</td>
</tr>
<tr>
<td>DRAWCURV, 3541</td>
<td>FFSLPE, 720</td>
</tr>
<tr>
<td>DRAWCX, 392</td>
<td>FFSQFR, 722</td>
</tr>
<tr>
<td>DRAWHACK, 397</td>
<td>FGLMICPK, 658</td>
</tr>
<tr>
<td>DRAWPNT, 3559</td>
<td>FLAGG2, 726</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>FLOATCP, 735</td>
<td></td>
</tr>
<tr>
<td>FLOATRP, 739</td>
<td></td>
</tr>
<tr>
<td>FOP, 747</td>
<td></td>
</tr>
<tr>
<td>FORDER, 662</td>
<td></td>
</tr>
<tr>
<td>FORMULA1, 3368</td>
<td></td>
</tr>
<tr>
<td>FORT, 750</td>
<td></td>
</tr>
<tr>
<td>FR2, 642</td>
<td></td>
</tr>
<tr>
<td>FRAC2, 771</td>
<td></td>
</tr>
<tr>
<td>FRIIDEAL2, 753</td>
<td></td>
</tr>
<tr>
<td>FRNAAF2, 773</td>
<td></td>
</tr>
<tr>
<td>FRUTIL, 644</td>
<td></td>
</tr>
<tr>
<td>FS2, 799</td>
<td></td>
</tr>
<tr>
<td>FS2EXXPXP, 814</td>
<td></td>
</tr>
<tr>
<td>FS2UPS, 826</td>
<td></td>
</tr>
<tr>
<td>FSAGG2, 733</td>
<td></td>
</tr>
<tr>
<td>FSCINT, 796</td>
<td></td>
</tr>
<tr>
<td>FSINT, 801</td>
<td></td>
</tr>
<tr>
<td>FSPECF, 775</td>
<td></td>
</tr>
<tr>
<td>FSPRMELT, 805</td>
<td></td>
</tr>
<tr>
<td>FSRAD, 809</td>
<td></td>
</tr>
<tr>
<td>FSUPFACT, 843</td>
<td></td>
</tr>
<tr>
<td>GALKACT, 852</td>
<td></td>
</tr>
<tr>
<td>GALFACTU, 848</td>
<td></td>
</tr>
<tr>
<td>GALPOLYU, 870</td>
<td></td>
</tr>
<tr>
<td>GALUTIL, 873</td>
<td></td>
</tr>
<tr>
<td>GAUSSFAC, 877</td>
<td></td>
</tr>
<tr>
<td>GB, 1002</td>
<td></td>
</tr>
<tr>
<td>GBEUCLID, 555</td>
<td></td>
</tr>
<tr>
<td>GBF, 957</td>
<td></td>
</tr>
<tr>
<td>GBININTERN, 965</td>
<td></td>
</tr>
<tr>
<td>GDRAW, 939</td>
<td></td>
</tr>
<tr>
<td>GENEZ, 924</td>
<td></td>
</tr>
<tr>
<td>GENMFACT, 886</td>
<td></td>
</tr>
<tr>
<td>GENPGCD, 904</td>
<td></td>
</tr>
<tr>
<td>GENUFAC, 929</td>
<td></td>
</tr>
<tr>
<td>GENUPS, 918</td>
<td></td>
</tr>
<tr>
<td>GHENSEL, 882</td>
<td></td>
</tr>
<tr>
<td>GOSPER, 942</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td></td>
</tr>
<tr>
<td>GRAY, 951</td>
<td></td>
</tr>
<tr>
<td>GRDEF, 948</td>
<td></td>
</tr>
<tr>
<td>GROEBSOL, 1006</td>
<td></td>
</tr>
<tr>
<td>GUESS, 1012</td>
<td></td>
</tr>
<tr>
<td>GUESSAN, 1048</td>
<td></td>
</tr>
<tr>
<td>GUESSF, 1050</td>
<td></td>
</tr>
<tr>
<td>GUESSF1, 1051</td>
<td></td>
</tr>
<tr>
<td>GUESST, 1053</td>
<td></td>
</tr>
<tr>
<td>GUESSP, 1055</td>
<td></td>
</tr>
<tr>
<td>GUESSUP, 1057</td>
<td></td>
</tr>
<tr>
<td>HB, 1064</td>
<td></td>
</tr>
<tr>
<td>HEUGCD, 1067</td>
<td></td>
</tr>
<tr>
<td>IALGFAC, 1099</td>
<td></td>
</tr>
<tr>
<td>IBACHIN, 188</td>
<td></td>
</tr>
<tr>
<td>IBATOOL, 1223</td>
<td></td>
</tr>
<tr>
<td>IBPTOOLS, 1227</td>
<td></td>
</tr>
<tr>
<td>ICDEN, 1102</td>
<td></td>
</tr>
<tr>
<td>IDECOMP, 1074</td>
<td></td>
</tr>
<tr>
<td>IMATLIN, 1105</td>
<td></td>
</tr>
<tr>
<td>IMATQF, 1111</td>
<td></td>
</tr>
<tr>
<td>INBFF, 1130</td>
<td></td>
</tr>
<tr>
<td>INCRMAPS, 1083</td>
<td></td>
</tr>
<tr>
<td>INE, 1139</td>
<td></td>
</tr>
<tr>
<td>INFINITY, 1097</td>
<td></td>
</tr>
<tr>
<td>INFORMI, 1159</td>
<td></td>
</tr>
<tr>
<td>INFPROD0, 1085</td>
<td></td>
</tr>
<tr>
<td>INFSP, 1144</td>
<td></td>
</tr>
<tr>
<td>INMODGCD, 1113</td>
<td></td>
</tr>
<tr>
<td>INNFACT, 1120</td>
<td></td>
</tr>
<tr>
<td>INPRODFF, 1087</td>
<td></td>
</tr>
<tr>
<td>INPRODPF, 1091</td>
<td></td>
</tr>
<tr>
<td>INPSIGN, 1149</td>
<td></td>
</tr>
<tr>
<td>INTAF, 31</td>
<td></td>
</tr>
<tr>
<td>INTALG, 23</td>
<td></td>
</tr>
<tr>
<td>INTBIT, 1164</td>
<td></td>
</tr>
<tr>
<td>INTDIVP, 1261</td>
<td></td>
</tr>
<tr>
<td>INTEF, 502</td>
<td></td>
</tr>
<tr>
<td>INTERGB, 1161</td>
<td></td>
</tr>
<tr>
<td>INFACT, 1172</td>
<td></td>
</tr>
<tr>
<td>INTFRSP, 1253</td>
<td></td>
</tr>
<tr>
<td>INTG0, 932</td>
<td></td>
</tr>
<tr>
<td>INTHET, 1197</td>
<td></td>
</tr>
<tr>
<td>INTHERAL, 20</td>
<td></td>
</tr>
<tr>
<td>INTHTR, 3564</td>
<td></td>
</tr>
<tr>
<td>INTPACK, 56</td>
<td></td>
</tr>
<tr>
<td>INTPAF, 3137</td>
<td></td>
</tr>
<tr>
<td>INTPM, 2908</td>
<td></td>
</tr>
<tr>
<td>INDEX Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>INVAPLA, 1275</td>
<td>MATLIN, 1520</td>
</tr>
<tr>
<td>IPRNTPK, 1246</td>
<td>MATSTOR, 3422</td>
</tr>
<tr>
<td>IR2, 1230</td>
<td>MCALCFN, 1602</td>
</tr>
<tr>
<td>IR2F, 1236</td>
<td>MCDEN, 1517</td>
</tr>
<tr>
<td>IROOT, 1216</td>
<td>MDFACT, 1556</td>
</tr>
<tr>
<td>IRREDFFX, 1264</td>
<td>MESH, 1552</td>
</tr>
<tr>
<td>IRRF2F, 1233</td>
<td>MFINFACT, 1589</td>
</tr>
<tr>
<td>IRSN, 1267</td>
<td>MHROWRED, 1562</td>
</tr>
<tr>
<td>IRURPK, 1248</td>
<td>MKBCFUNC, 1454</td>
</tr>
<tr>
<td>ISUMP, 1151</td>
<td>MKFLCFN, 1456</td>
</tr>
<tr>
<td>ITFUN2, 1093</td>
<td>MKFUNC, 1464</td>
</tr>
<tr>
<td>ITFUN3, 1095</td>
<td>MKRECORD, 1466</td>
</tr>
<tr>
<td>ITRIGMNP, 1154</td>
<td>MKUCFUNC, 1468</td>
</tr>
<tr>
<td>KERNEL2, 1282</td>
<td>MMAP, 1600</td>
</tr>
<tr>
<td>KOVACIC, 1284</td>
<td>MONOTOOL, 1570</td>
</tr>
<tr>
<td>LAPLACE, 1288</td>
<td>MPC2, 1582</td>
</tr>
<tr>
<td>LAZM3P, 1313</td>
<td>MPC3, 1584</td>
</tr>
<tr>
<td>LEADCDET, 1317</td>
<td>MPCPF, 1575</td>
</tr>
<tr>
<td>LEXTRIPK, 1390</td>
<td>MPRFF, 1578</td>
</tr>
<tr>
<td>LF, 1432</td>
<td>MRATFAC, 1586</td>
</tr>
<tr>
<td>LGROBP, 1420</td>
<td>MRF2, 1568</td>
</tr>
<tr>
<td>LIMITPS, 3077</td>
<td>MSYSCMD, 1573</td>
</tr>
<tr>
<td>LIMITRF, 3225</td>
<td>MTHING, 1550</td>
</tr>
<tr>
<td>LINDEP, 1396</td>
<td>MULTFACT, 1607</td>
</tr>
<tr>
<td>LIST2, 1437</td>
<td>MULTSQFR, 1615</td>
</tr>
<tr>
<td>LIST2MAP, 1442</td>
<td>NAGC02, 2451</td>
</tr>
<tr>
<td>LIST3, 1440</td>
<td>NAGC05, 2470</td>
</tr>
<tr>
<td>LISYSER, 1409</td>
<td>NAGC06, 2518</td>
</tr>
<tr>
<td>LODEEF, 474</td>
<td>NAGD01, 1999</td>
</tr>
<tr>
<td>LODOF, 1399</td>
<td>NAGD02, 2386</td>
</tr>
<tr>
<td>LODOOPS, 1403</td>
<td>NAGD03, 2433</td>
</tr>
<tr>
<td>LOP, 1428</td>
<td>NAGE01, 2047</td>
</tr>
<tr>
<td>LPARSPT, 1445</td>
<td>NAGE02, 1832</td>
</tr>
<tr>
<td>LPEFRAC, 1406</td>
<td>NAGE04, 2287</td>
</tr>
<tr>
<td>LSMP, 1412</td>
<td>NAGF01, 2128</td>
</tr>
<tr>
<td>LSMP1, 1415</td>
<td>NAGF02, 1690</td>
</tr>
<tr>
<td>LSPP, 1418</td>
<td>NAGF04, 1910</td>
</tr>
<tr>
<td>MAMA, 1542</td>
<td>NAGF07, 2068</td>
</tr>
<tr>
<td>MAPHACK1, 1470</td>
<td>NAGS, 2671</td>
</tr>
<tr>
<td>MAPHACK2, 1472</td>
<td>NAGSP, 1920</td>
</tr>
<tr>
<td>MAPHACK3, 1474</td>
<td>NCEP, 2774</td>
</tr>
<tr>
<td>MAPPKG1, 1484</td>
<td>NCNTFRAC, 2777</td>
</tr>
<tr>
<td>MAPPKG2, 1494</td>
<td>NCODIV, 2698</td>
</tr>
<tr>
<td>MAPPKG3, 1505</td>
<td>NEWTON, 2690</td>
</tr>
<tr>
<td>MAPPKG4, 1512</td>
<td>NFINTBAS, 2726</td>
</tr>
<tr>
<td>MATCAT2, 1512</td>
<td>NLINSOL, 2708</td>
</tr>
<tr>
<td>INDEX</td>
<td>3873</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>POLYCATQ, 2997</td>
<td></td>
</tr>
<tr>
<td>POLYLIFT, 2995</td>
<td></td>
</tr>
<tr>
<td>POLYROOT, 3042</td>
<td></td>
</tr>
<tr>
<td>POLYVEC, 3704</td>
<td></td>
</tr>
<tr>
<td>PREASSOC, 3089</td>
<td></td>
</tr>
<tr>
<td>PRIMARR2, 3092</td>
<td></td>
</tr>
<tr>
<td>PRIMELT, 3095</td>
<td></td>
</tr>
<tr>
<td>PRIMES, 1203</td>
<td></td>
</tr>
<tr>
<td>PRINT, 3110</td>
<td></td>
</tr>
<tr>
<td>PRJALGPK, 2961</td>
<td></td>
</tr>
<tr>
<td>PRS, 3116</td>
<td></td>
</tr>
<tr>
<td>PSETPK, 3046</td>
<td></td>
</tr>
<tr>
<td>PSEUDLIN, 3112</td>
<td></td>
</tr>
<tr>
<td>PSQFR, 3071</td>
<td></td>
</tr>
<tr>
<td>PTFUNC2, 2966</td>
<td></td>
</tr>
<tr>
<td>PTPACK, 2968</td>
<td></td>
</tr>
<tr>
<td>PUSUVAR, 3148</td>
<td></td>
</tr>
<tr>
<td>PWFFINTB, 2853</td>
<td></td>
</tr>
<tr>
<td>QALGSET2, 3152</td>
<td></td>
</tr>
<tr>
<td>QCMPACK, 3156</td>
<td></td>
</tr>
<tr>
<td>QFCAT2, 3165</td>
<td></td>
</tr>
<tr>
<td>QUATCT2, 3168</td>
<td></td>
</tr>
<tr>
<td>RADUTIL, 3197</td>
<td></td>
</tr>
<tr>
<td>RANDSRC, 3207</td>
<td></td>
</tr>
<tr>
<td>RATFACT, 3209</td>
<td></td>
</tr>
<tr>
<td>RATRET, 3254</td>
<td></td>
</tr>
<tr>
<td>RDEEF, 512</td>
<td></td>
</tr>
<tr>
<td>RDEEFS, 521</td>
<td></td>
</tr>
<tr>
<td>RDETR, 3592</td>
<td></td>
</tr>
<tr>
<td>RDETRS, 3597</td>
<td></td>
</tr>
<tr>
<td>RDIST, 3199</td>
<td></td>
</tr>
<tr>
<td>RDIV, 3301</td>
<td></td>
</tr>
<tr>
<td>REAL0, 3277</td>
<td></td>
</tr>
<tr>
<td>REALOQ, 3284</td>
<td></td>
</tr>
<tr>
<td>REALSOLV, 3274</td>
<td></td>
</tr>
<tr>
<td>RECPOR, 3289</td>
<td></td>
</tr>
<tr>
<td>REORDER, 3306</td>
<td></td>
</tr>
<tr>
<td>REP, 3172</td>
<td></td>
</tr>
<tr>
<td>REP1, 3330</td>
<td></td>
</tr>
<tr>
<td>REP2, 3338</td>
<td></td>
</tr>
<tr>
<td>REPDB, 3325</td>
<td></td>
</tr>
<tr>
<td>REPSEQ, 3328</td>
<td></td>
</tr>
<tr>
<td>RESLATIC, 3355</td>
<td></td>
</tr>
<tr>
<td>RESSOL, 3357</td>
<td></td>
</tr>
<tr>
<td>RF, 3212</td>
<td></td>
</tr>
<tr>
<td>RFDIST, 3201</td>
<td></td>
</tr>
<tr>
<td>RFFACT, 3218</td>
<td></td>
</tr>
<tr>
<td>RFFACTOR, 3220</td>
<td></td>
</tr>
<tr>
<td>RFP, 3360</td>
<td></td>
</tr>
<tr>
<td>RIDIST, 3204</td>
<td></td>
</tr>
<tr>
<td>RINTERP, 3244</td>
<td></td>
</tr>
<tr>
<td>RMCAT2, 3287</td>
<td></td>
</tr>
<tr>
<td>RSDCMPK, 3310</td>
<td></td>
</tr>
<tr>
<td>RSETGCD, 3317</td>
<td></td>
</tr>
<tr>
<td>RURPK, 3263</td>
<td></td>
</tr>
<tr>
<td>SAEFACT, 3374</td>
<td></td>
</tr>
<tr>
<td>SAERFFC, 3366</td>
<td></td>
</tr>
<tr>
<td>SCACHE, 3384</td>
<td></td>
</tr>
<tr>
<td>SCPKG, 3466</td>
<td></td>
</tr>
<tr>
<td>SEG2, 3372</td>
<td></td>
</tr>
<tr>
<td>SEGBIND2, 3370</td>
<td></td>
</tr>
<tr>
<td>SFQCMPPK, 3394</td>
<td></td>
</tr>
<tr>
<td>SFRGCD, 3411</td>
<td></td>
</tr>
<tr>
<td>SGCIF, 3498</td>
<td></td>
</tr>
<tr>
<td>SHP, 3470</td>
<td></td>
</tr>
<tr>
<td>SIGNEDF, 487</td>
<td></td>
</tr>
<tr>
<td>SIGNRF, 3230</td>
<td></td>
</tr>
<tr>
<td>SIMPAN, 3376</td>
<td></td>
</tr>
<tr>
<td>SMITH, 3378</td>
<td></td>
</tr>
<tr>
<td>SOLIZEF, 3064</td>
<td></td>
</tr>
<tr>
<td>SOLVERAD, 3189</td>
<td></td>
</tr>
<tr>
<td>SOLVESER, 3620</td>
<td></td>
</tr>
<tr>
<td>SOLVETRA, 3608</td>
<td></td>
</tr>
<tr>
<td>SORTPAK, 3387</td>
<td></td>
</tr>
<tr>
<td>SPECOUT, 3391</td>
<td></td>
</tr>
<tr>
<td>SRDCMPK, 3404</td>
<td></td>
</tr>
<tr>
<td>STINPROD, 3434</td>
<td></td>
</tr>
<tr>
<td>STNSR, 3447</td>
<td></td>
</tr>
<tr>
<td>STREAM1, 3427</td>
<td></td>
</tr>
<tr>
<td>STREAM2, 3429</td>
<td></td>
</tr>
<tr>
<td>STREAM3, 3432</td>
<td></td>
</tr>
<tr>
<td>STTAYLOR, 3437</td>
<td></td>
</tr>
<tr>
<td>STTTF, 3449</td>
<td></td>
</tr>
<tr>
<td>STTTFNC, 3460</td>
<td></td>
</tr>
<tr>
<td>SUBRESP, 3479</td>
<td></td>
</tr>
<tr>
<td>SUMFS, 811</td>
<td></td>
</tr>
<tr>
<td>SUMRF, 3238</td>
<td></td>
</tr>
<tr>
<td>SUP2, 3389</td>
<td></td>
</tr>
<tr>
<td>SUPFRAVE, 3483</td>
<td></td>
</tr>
<tr>
<td>SYMFUNC, 3509</td>
<td></td>
</tr>
<tr>
<td>SYSSOLP, 3492</td>
<td></td>
</tr>
<tr>
<td>TABLBUFP, 3512</td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>TANEXP, 3520</td>
<td></td>
</tr>
<tr>
<td>TBCMPPK, 3516</td>
<td></td>
</tr>
<tr>
<td>TEMUTL, 3527</td>
<td></td>
</tr>
<tr>
<td>TEX1, 3529</td>
<td></td>
</tr>
<tr>
<td>TOOLSIGN, 3531</td>
<td></td>
</tr>
<tr>
<td>TOPSP, 3562</td>
<td></td>
</tr>
<tr>
<td>TRIGMNIP, 3627</td>
<td></td>
</tr>
<tr>
<td>TRIMAT, 3624</td>
<td></td>
</tr>
<tr>
<td>TRMANIP, 3577</td>
<td></td>
</tr>
<tr>
<td>TUBETOOL, 3631</td>
<td></td>
</tr>
<tr>
<td>TWOFACT, 3642</td>
<td></td>
</tr>
<tr>
<td>UDPO, 3696</td>
<td></td>
</tr>
<tr>
<td>UDVO, 3699</td>
<td></td>
</tr>
<tr>
<td>UFPS1, 3658</td>
<td></td>
</tr>
<tr>
<td>ULS2, 3660</td>
<td></td>
</tr>
<tr>
<td>UNIFACT, 3650</td>
<td></td>
</tr>
<tr>
<td>UNISEG2, 3694</td>
<td></td>
</tr>
<tr>
<td>UP2, 3672</td>
<td></td>
</tr>
<tr>
<td>UPCDEN, 3664</td>
<td></td>
</tr>
<tr>
<td>UPDECOMP, 3666</td>
<td></td>
</tr>
<tr>
<td>UPDIVP, 3670</td>
<td></td>
</tr>
<tr>
<td>UPMP, 3674</td>
<td></td>
</tr>
<tr>
<td>UPOLYC2, 3662</td>
<td></td>
</tr>
<tr>
<td>UPSQFREE, 3678</td>
<td></td>
</tr>
<tr>
<td>UPXS2, 3681</td>
<td></td>
</tr>
<tr>
<td>UTS2, 3688</td>
<td></td>
</tr>
<tr>
<td>UTSODE, 3690</td>
<td></td>
</tr>
<tr>
<td>UTSODET, 3701</td>
<td></td>
</tr>
<tr>
<td>UTSSOL, 3523</td>
<td></td>
</tr>
<tr>
<td>VECTOR2, 3718</td>
<td></td>
</tr>
<tr>
<td>VIEW, 3726</td>
<td></td>
</tr>
<tr>
<td>VIEWDEF, 3720</td>
<td></td>
</tr>
<tr>
<td>WEIER, 3732</td>
<td></td>
</tr>
<tr>
<td>WFFINTBS, 3736</td>
<td></td>
</tr>
<tr>
<td>XEXPPKG, 3744</td>
<td></td>
</tr>
<tr>
<td>YSTREAM, 2859</td>
<td></td>
</tr>
<tr>
<td>ZDSOLVE, 3812</td>
<td></td>
</tr>
<tr>
<td>ZLINDEP, 1182</td>
<td></td>
</tr>
<tr>
<td>IMATLIN, 1105</td>
<td></td>
</tr>
<tr>
<td>IMATQF, 1111</td>
<td></td>
</tr>
<tr>
<td>inverse, 1111</td>
<td></td>
</tr>
<tr>
<td>nullSpace, 1111</td>
<td></td>
</tr>
<tr>
<td>rowEchelon, 1111</td>
<td></td>
</tr>
<tr>
<td>inconsistent?</td>
<td></td>
</tr>
<tr>
<td>PLEQN, 2862</td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>INCRMAPS, 1083</td>
<td></td>
</tr>
<tr>
<td>incrementBy</td>
<td></td>
</tr>
<tr>
<td>INCRMAPS, 1083</td>
<td></td>
</tr>
<tr>
<td>IncrementingMaps, 1083</td>
<td></td>
</tr>
<tr>
<td>INCRMAPS, 1083</td>
<td></td>
</tr>
<tr>
<td>index</td>
<td></td>
</tr>
<tr>
<td>INBFF, 1130</td>
<td></td>
</tr>
<tr>
<td>indiceSubResultant</td>
<td></td>
</tr>
<tr>
<td>PRS, 3117</td>
<td></td>
</tr>
<tr>
<td>indiceSubResultantEuclidean</td>
<td></td>
</tr>
<tr>
<td>PRS, 3117</td>
<td></td>
</tr>
<tr>
<td>indicialEquation</td>
<td></td>
</tr>
<tr>
<td>ODEPRIM, 3098</td>
<td></td>
</tr>
<tr>
<td>indicialEquationAtInfinity</td>
<td></td>
</tr>
<tr>
<td>ODERAT, 3248</td>
<td></td>
</tr>
<tr>
<td>indicialEquations</td>
<td></td>
</tr>
<tr>
<td>ODEPRIM, 3098</td>
<td></td>
</tr>
<tr>
<td>INEP, 1138</td>
<td></td>
</tr>
<tr>
<td>charpol, 1139</td>
<td>DTP, 303</td>
</tr>
<tr>
<td>innerEigenvalues, 1139</td>
<td>INMODGCD, 1113</td>
</tr>
<tr>
<td>solve1, 1139</td>
<td>modularGcd, 1113</td>
</tr>
<tr>
<td>infIELDINT</td>
<td>modularGcdPrimitive, 1113</td>
</tr>
<tr>
<td>INTRAT, 3241</td>
<td>reduction, 1113</td>
</tr>
<tr>
<td>infIELDIntegrate</td>
<td>InnerAlgFactor, 1099</td>
</tr>
<tr>
<td>INTRF, 3223</td>
<td>InnerCommonDenominator, 1102</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>innerEigenvalues</td>
</tr>
<tr>
<td>INFPRODO, 1085</td>
<td>INEP, 1139</td>
</tr>
<tr>
<td>INFPRODF, 1087</td>
<td>innerint</td>
</tr>
<tr>
<td>INFPRODF, 1091</td>
<td>DEFINTEF, 468</td>
</tr>
<tr>
<td>STINPROD, 3434</td>
<td>InnerMatrixLinearAlgebraFunctions, 1105</td>
</tr>
<tr>
<td>InfiniteProductCharacteristicZero, 1085</td>
<td>InnerMatrixQuotientFieldFunctions, 1111</td>
</tr>
<tr>
<td>InfiniteProductFiniteField, 1087</td>
<td>InnerModularGcd, 1113</td>
</tr>
<tr>
<td>InfiniteProductPrimeField, 1091</td>
<td>InnerMultFact, 1120</td>
</tr>
<tr>
<td>InfiniteTupleFunctions2, 1093</td>
<td>InnerNormalBasisFieldFunctions, 1130</td>
</tr>
<tr>
<td>InfiniteTupleFunctions3, 1095</td>
<td>InnerNumericEigenPackage, 1138</td>
</tr>
<tr>
<td>INFINITY, 1097</td>
<td>InnerNumericFloatSolvePackage, 1144</td>
</tr>
<tr>
<td>infinity, 1097</td>
<td>InnerPolySign, 1149</td>
</tr>
<tr>
<td>minusInfinity, 1097</td>
<td>InnerPolySum, 1151</td>
</tr>
<tr>
<td>plusInfinity, 1097</td>
<td>innerSolve</td>
</tr>
<tr>
<td>Infinity, 1097</td>
<td>INFSP, 1144</td>
</tr>
<tr>
<td>infinity</td>
<td>innerSolve1</td>
</tr>
<tr>
<td>infinityNorm</td>
<td>INFSP, 1144</td>
</tr>
<tr>
<td>GALFACTU, 848</td>
<td>InnerTrigonometricManipulations, 1154</td>
</tr>
<tr>
<td>INFORM1, 1159</td>
<td>INNMFACT, 1120</td>
</tr>
<tr>
<td>interpret, 1159</td>
<td>factor, 1120</td>
</tr>
<tr>
<td>packageCall, 1159</td>
<td>INFPRODO, 1085</td>
</tr>
<tr>
<td>evenInfiniteProduct, 1085</td>
<td>evenInfiniteProduct, 1087</td>
</tr>
<tr>
<td>generalInfiniteProduct, 1085</td>
<td>generalInfiniteProduct, 1087</td>
</tr>
<tr>
<td>infiniteProduct, 1085</td>
<td>infiniteProduct, 1087</td>
</tr>
<tr>
<td>oddInfiniteProduct, 1085</td>
<td>oddInfiniteProduct, 1087</td>
</tr>
<tr>
<td>infRittWu?</td>
<td>INFPRODF, 1091</td>
</tr>
<tr>
<td>QCMPACK, 3156</td>
<td>evenInfiniteProduct, 1091</td>
</tr>
<tr>
<td>SFQCMPK, 3394</td>
<td>generalInfiniteProduct, 1091</td>
</tr>
<tr>
<td>INFSP, 1144</td>
<td>infiniteProduct, 1091</td>
</tr>
<tr>
<td>innerSolve, 1144</td>
<td>oddInfiniteProduct, 1091</td>
</tr>
<tr>
<td>innerSolve1, 1144</td>
<td>INPSIGN, 1149</td>
</tr>
<tr>
<td>makeEq, 1144</td>
<td>signAround, 1149</td>
</tr>
<tr>
<td>inHALLBasis?</td>
<td>InputFormFunctions1, 1159</td>
</tr>
<tr>
<td>HB, 1064</td>
<td>inRootof</td>
</tr>
<tr>
<td>initializeParamOfPlaces</td>
<td>AF, 14</td>
</tr>
<tr>
<td>DTP, 303</td>
<td>int</td>
</tr>
<tr>
<td>initParLocLeaves</td>
<td>ODEINT, 2790</td>
</tr>
<tr>
<td></td>
<td>STTAYLOR, 3437</td>
</tr>
<tr>
<td></td>
<td>INTAF, 31</td>
</tr>
</tbody>
</table>
INDEX

algint, 31
INTALG, 23
algintegrate, 23
palginfeldint, 23
palgintegrate, 23
INTBIT, 1164
bitCoef, 1164
bitLength, 1164
bitTruth, 1164
intChoose
MULTSQFR, 1615
intcompBasis
LGROBP, 1420
INTDIVP, 1261
intersectionDivisor, 1261
placesOfDegree, 1261
INTEF, 502
lfextend dint, 502
lfextlint, 502
lfinfieldint, 502
lfintegrate, 502
lflimitedint, 502
integer
INTRET, 1214
integer?
INTRET, 1214
IntegerBits, 1164
integerBound
BOUNDZRO, 170
IntegerCombinatoricFunctions, 1167
IntegerFactorizationPackage, 1172
integerIfCan
INTRET, 1214
IntegerLinearDependence, 1182
IntegerNumberTheoryFunctions, 1197
IntegerPrimesPackage, 1203
IntegerRetractions, 1214
IntegerRoots, 1216
integers
STTAYLOR, 3437
IntegerSolveLinearPolynomialEquation, 1221
integral
LF, 1432
integralBasis
FFINTBAS, 787
NFINTBAS, 2726
PWFFINTB, 2853
WFFINTBS, 3736
IntegralBasisPolynomialTools, 1227
IntegralBasisTools, 1223
integralLastSubResultant
RSETGCD, 3317
integrate
DEFINTEF, 468
DEFINTRF, 3215
FSINT, 801
INTPACK, 56
INTRAT, 3241
IRRF2F, 1233
STTAYLOR, 3437
IntegrationResultFunctions2, 1230
IntegrationResultRFToFunction, 1233
IntegrationResultToFunction, 1236
IntegrationTools, 1242
InterfaceGroebnerPackage, 1161
INTERGB, 1161
groebner, 1161
intermediateResultsIF
D02AGNT, 420
internalDecompose
RSDCMPK, 3310
SRDCMPK, 3404
internalInfRittWu?
QCMPACK, 3156
SFQCMPK, 3394
internalIntegrate
FSCINT, 796
INTRF, 3223
internalIntegrate0
FSCINT, 796
InternalPrintPackage, 1246
InternalRationalUnivariateRepresentationPackage, 1248
internalSubPolSet?
QCMPACK, 3156
SFQCMPK, 3394
internalSubQuasiComponent?
QCMPACK, 3156
SFQCMPK, 3394
interpolate
FFFG, 756
PINTERP, 3033
<table>
<thead>
<tr>
<th>INDEX</th>
<th>3877</th>
</tr>
</thead>
<tbody>
<tr>
<td>RINTERP, 3244</td>
<td>divisors, 1197</td>
</tr>
<tr>
<td>interpolateForms</td>
<td>euler, 1197</td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td>eulerPhi, 1197</td>
</tr>
<tr>
<td>INTFRSP, 1253</td>
<td>fibonacci, 1197</td>
</tr>
<tr>
<td>PAFF, 2821</td>
<td>harmonic, 1197</td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td>jacob, 1197</td>
</tr>
<tr>
<td>interpolateFormsForFact</td>
<td>legendre, 1197</td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td>moebiusMu, 1197</td>
</tr>
<tr>
<td>INTFRSP, 1253</td>
<td>numberOfDivisors, 1197</td>
</tr>
<tr>
<td>PAFF, 2821</td>
<td>sumOfDivisors, 1197</td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td>sumOfKthPowerDivisors, 1197</td>
</tr>
<tr>
<td>InterpolateFormsPackage, 1253</td>
<td>INTHERAL, 19</td>
</tr>
<tr>
<td>interpret</td>
<td>HermiteIntegrate, 20</td>
</tr>
<tr>
<td>INFORM1, 1159</td>
<td>INTHERTR, 3564</td>
</tr>
<tr>
<td>interpretString</td>
<td>HermiteIntegrate, 3564</td>
</tr>
<tr>
<td>TEMUTL, 3527</td>
<td>INTPACK, 56</td>
</tr>
<tr>
<td>interReduce</td>
<td>integrate, 56</td>
</tr>
<tr>
<td>PSETPK, 3047</td>
<td>measure, 56</td>
</tr>
<tr>
<td>intersectionDivisor</td>
<td>INTPAF, 3137</td>
</tr>
<tr>
<td>GPAFF, 889</td>
<td>palgextint, 3137</td>
</tr>
<tr>
<td>INTDIVP, 1261</td>
<td>palgint, 3137</td>
</tr>
<tr>
<td>PAFF, 2821</td>
<td>palglimint, 3137</td>
</tr>
<tr>
<td>PAFFFF, 2829</td>
<td>palgLODE, 3137</td>
</tr>
<tr>
<td>InterpolationDivisorPackage, 1261</td>
<td>palgRDE, 3137</td>
</tr>
<tr>
<td>INTFACT, 1172</td>
<td>intPatternMatch</td>
</tr>
<tr>
<td>BasicMethod, 1172</td>
<td>INTTOOLS, 1242</td>
</tr>
<tr>
<td>factor, 1172</td>
<td>INTPM, 2908</td>
</tr>
<tr>
<td>PollardSmallFactor, 1172</td>
<td>pmComplexintegrate, 2908</td>
</tr>
<tr>
<td>squareFree, 1172</td>
<td>pmintegrate, 2908</td>
</tr>
<tr>
<td>INTFRSP, 1253</td>
<td>splitConstant, 2908</td>
</tr>
<tr>
<td>basisOfInterpolateForms, 1253</td>
<td>INTRAT, 3241</td>
</tr>
<tr>
<td>basisOfInterpolateFormsForFact, 1253</td>
<td>extendedint, 3241</td>
</tr>
<tr>
<td>interpolateForms, 1253</td>
<td>infieldint, 3241</td>
</tr>
<tr>
<td>interpolateFormsForFact, 1253</td>
<td>integrate, 3241</td>
</tr>
<tr>
<td>INTG0, 932</td>
<td>limitedint, 3241</td>
</tr>
<tr>
<td>lift, 932</td>
<td>INTRET, 1214</td>
</tr>
<tr>
<td>multivariate, 932</td>
<td>integer, 1214</td>
</tr>
<tr>
<td>palgextint0, 932</td>
<td>integer?, 1214</td>
</tr>
<tr>
<td>palgint0, 932</td>
<td>integerIfCan, 1214</td>
</tr>
<tr>
<td>palglimint0, 932</td>
<td>INTRF, 3223</td>
</tr>
<tr>
<td>palgLODE0, 932</td>
<td>extendedIntegrate, 3223</td>
</tr>
<tr>
<td>palgRDE0, 932</td>
<td>infieldIntegrate, 3223</td>
</tr>
<tr>
<td>univariate, 932</td>
<td>internalIntegrate, 3223</td>
</tr>
<tr>
<td>INTHEORY, 1197</td>
<td>limitedIntegrate, 3223</td>
</tr>
<tr>
<td>bernoulli, 1197</td>
<td>INTSLPE, 1221</td>
</tr>
<tr>
<td>chineseRemainder, 1197</td>
<td>solveLinearPolynomialEquation, 1221</td>
</tr>
</tbody>
</table>
INTTOOLS, 1242
  intPatternMatch, 1242
kmax, 1242
ksec, 1242
mkPrim, 1242
removeConstantTerm, 1242
union, 1242
vark, 1242
varselect, 1242

INTTR, 3567
  expextendedint, 3567
  expintegrate, 3567
  expintfdpoly, 3567
  explimitedint, 3567
  monomialIntegrate, 3567
  monomialIntPoly, 3567
  primextendedint, 3567
  primextintfrac, 3567
  primintegrate, 3567
  primintfdpoly, 3567
  primlimitintfrac, 3567
  primlimitedint, 3567
  tanintegrate, 3567

inv
  INBFF, 1130
inverse
  IMATQF, 1111
  MATLIN, 1520
  TABLBUMP, 3512
inverseColeman
  SGCF, 3498
inverseLaplace
  INVLAPLA, 1275
  InverseLaplaceTransform, 1275
invertIfCan
  MATLIN, 1520
INVLAGPLA, 1275
  inverseLaplace, 1275
invmultisect
  STTAYLOR, 3437
ipow
  COMBF, 198
iprint
  IPRNTPK, 1246
IR2, 1230
  complexExpand, 1236
  expand, 1236
  split, 1236
IR2F, 1236
  complexExpand, 1236
  expand, 1236
  split, 1236
IROOT, 1216
  approxNthRoot, 1216
  approxSqrt, 1216
  perfectNthPower?, 1216
  perfectNthRoot, 1216
  perfectSqrt, 1216
  perfectSquare?, 1216
irroot
  AF, 14
IRREDFFX, 1264
  generateIrredPoly, 1264
IrredPolyOverFiniteField, 1264
irreducible?
  DDFACT, 328
  FFFACTOR, 668
  FFFACTSE, 674
irreducibleFactors
  PSETPK, 3047
irreducibleRepresentation
  IRSN, 1267
IRRF2F, 1233
  complexExpand, 1233
  complexIntegrate, 1233
  expand, 1233
  integrate, 1233
  split, 1233
IrrRepSymNatPackage, 1267
IRSN, 1267
  dimensionOfIrreducibleRepresentation, 1267
  irreducibleRepresentation, 1267
IRURPK, 1248
  checkRur, 1248
  rur, 1248
Is
  PATMATCH, 2897
is?
  PATHMATCH, 2897
isAbsolutelyIrreducible?
  REP2, 3338
isExpt
INDEX

PSETPK, 3047
lazyVariations
POLUTIL, 3268
l Basis
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
lcm
POLYVEC, 3704
ldf2lst
D01AGNT, 406
ESCONT, 569
ESTOOLS, 578
ldf2vmf
ESTOOLS, 578
LEADCDET, 1317
distFact, 1317
polCase, 1317
LeadingCoefficientDetermination, 1317
leadingCoefficient, 3025
leadingCoefficientRicDE
ODEPRRIC, 3103
leastAffineMultiple
FFPOLY, 695
leastPower
IBATOOL, 1223
leftDivide
NCODIV, 2698
OREPCTO, 3684
leftExactQuotient
NCODIV, 2698
leftFactor
PDECOMP, 3003
leftFactorIfCan
UPDECOMP, 3666
leftGcd
NCODIV, 2698
leftLcm
NCODIV, 2698
leftQuotient
NCODIV, 2698
leftRank
ALGPKG, 42
leftRemainder
NCODIV, 2698
legendre
INTHEORY, 1197
PNTHEORY, 3037
legendreP
ORTHPOL, 2813
length
GALFACTU, 848
lepol
GBINTERN, 966
less?
UDPO, 3696
lex
TABLBUMP, 3512
lexGroebner
PGROEB, 2990
lexTriangular
LEXTRIPK, 1390
LexTriangularPackage, 1390
LEXTRIPK, 1390
glmIfCan, 1390
groebner, 1390
lexTriangular, 1390
squareFreeLexTriangular, 1390
zeroDimensional?, 1390
zeroSetSplit, 1390
LF, 1432
belong?, 1432
Ci, 1432
dilog, 1432
Ei, 1432
erf, 1432
integral, 1432
li, 1432
operator, 1432
Si, 1432
lfextendedint
INTEF, 502
lfextlimint
INTEF, 502
lfinfiefdint
INTEF, 502
lfintegrate
INTEF, 502
lflimitedint
INTEF, 502
lffunc
HB, 1064
INDEX

LGROBP, 1420
  anticoord, 1420
  choosemon, 1420
  computeBasis, 1420
  coord, 1420
  groebgen, 1420
  intcompBasis, 1420
  linGenPos, 1420
  minPol, 1420
  totolex, 1420
  transform, 1420
lin
  LF, 1432
lift
  INTG0, 932
  MULTSQFR, 1615
limit
  LIMITPS, 3077
  LIMITRF, 3225
limitedint
  INTRAT, 3241
limitedIntegrate
  INTRF, 3223
LIMITPS, 3077
  complexLimit, 3077
  limit, 3077
LIMITRF, 3225
  complexLimit, 3225
  limit, 3225
LINDEP, 1396
  linearDependence, 1396
  linearlyDependent?, 1396
  solveLinear, 1396
linear
  SOLVEFOR, 3065
linear?
  E04AGNT, 629
  PSETPK, 3047
linearBezier
  BEZIER, 157
LinearDependence, 1396
linearDependence
  LINDEP, 1396
linearDependenceOverZ
  ZLINDEP, 1182
linearlyDependent?
  LINDEP, 1396
  linearlyDependentOverZ?
    ZLINDEP, 1182
linearMatrix
  E04AGNT, 629
LinearOrdinaryDifferentialOperatorFactorizer, 1399
LinearOrdinaryDifferentialOperatorsOps, 1403
linearPart
  E04AGNT, 629
LinearPolynomialEquationByFractions, 1406
linearPolynomials
  PSETPK, 3047
linears
  MDDFACT, 1556
LinearSystemFromPowerSeriesPackage, 1409
LinearSystemMatrixPackage, 1412
LinearSystemMatrixPackage1, 1415
LinearSystemPolynomialPackage, 1418
lineColorDefault
  VIEWDEF, 3720
LinesOpPack, 1428
linGenPos
  LGROBP, 1420
LinGroebnerPackage, 1420
linkToFortran
  FORT, 750
linSolve
  LSPP, 1418
lintgcd
  HEUGCD, 1067
LiouvilianFunction, 1432
LIST2, 1437
  map, 1437
  reduce, 1437
  scan, 1437
LIST2MAP, 1442
  match, 1442
LIST3, 1440
  map, 1440
listAllMono
  PFORP, 2838
listAllMonoExp
  PFORP, 2838
listConjugateBases
  IBACHIN, 188
INDEX

listexp
   NPCOEF, 2721
ListFunctions2, 1437
ListFunctions3, 1440
ListToMap, 1442
listVariable
   PFORP, 2838
listYoungTableaus
   SGCF, 3498
LISYSER, 1409
   finiteSeries2LinSys, 1409
   finiteSeries2LinSysWOVectorise, 1409
   finiteSeries2Vector, 1409
localAbs
   FS2EXPXP, 814
   FS2UPS, 826
localIntegralBasis
   FFINTBAS, 787
   NFINTBAS, 2726
   PWFFINTB, 2853
   WFFINTBS, 3736
localize
   LPARSPT, 1445
   LocalParametrizationOfSimplePointPackage, 1445
localParamOfSimplePt
   LPARSPT, 1445
localReal?
   EF, 450
localUnquote
   APPRULE, 104
LODEEF, 474
   solve, 474
LODO2FUN
   UTSODETL, 3701
LODOF, 1399
   factor, 1399
   factor1, 1399
LODOOPS, 1403
   directSum, 1403
   symmetricPower, 1403
   symmetricProduct, 1403
log
   EF, 450
   FACTFUNC, 638
   XEXPPKG, 3744
logGamma
   DFSFUN, 348
lookup
   INBFF, 1130
loopPoints
   TUBETOOL, 3631
LOP, 1428
   quotVecSpaceBasis, 1428
   reduceLineOverLine, 1428
   reduceRow, 1428
   reduceRowOnList, 1428
   rowEchWoZeroLines, 1428
   rowEchWoZeroLinesWOVectorise, 1428
lowerPolynomial
   FACUTIL, 647
LowTriBddDenomInv
   TRIMAT, 3624
LPARSPT, 1445
   localize, 1445
   localParamOfSimplePt, 1445
   pointDominateBy, 1445
   pointToPlace, 1445
   printInfo, 1445
LPEFRAC, 1406
   solveLinearPolynomialEquationByFractions, 1406
LPolynomial
   GPAFF, 889
   PAFF, 2821
   PAFFFF, 2829
LSMP, 1412
   hasSolution?, 1412
   particularSolution, 1412
   rank, 1412
   solve, 1412
LSMP1, 1415
   hasSolution?, 1415
   particularSolution, 1415
   rank, 1415
   solve, 1415
LSPP, 1418
   linSolve, 1418
mainVariable
   POLYCATQ, 2997
   RF, 3212
INDEX

MakeBinaryCompiledFunction, 1454
makeCrit
    GBINTERN, 966
makeEq
    INFSP, 1144
MakeFloatCompiledFunction, 1456
makeFR
    GALFACT, 853
MakeFunction, 1464
makeObject
    DRAW, 3533
    DRAWCFUN, 3545
MakeRecord, 1466
makeRecord
    MKRECORD, 1466
MakeUnaryCompiledFunction, 1468
makeYoungTableau
    SGCF, 3498
makingStats?
    TBCMPPK, 3516
MAMA, 1542
    aColumn, 1543
    aRow, 1543
    bandMatrix, 1543
    blockConcat, 1543
    blockSplit, 1543
    columns, 1543
diagonalMatrix, 1543
element, 1543
horizConcat, 1543
horizSplit, 1543
    rows, 1543
    subMatrix, 1543
vertConcat, 1543
vertSplit, 1543
map
    ARRAY12, 2796
    CARTEN2, 178
    COMPLEX2, 224
    DIRPROD2, 318
    EQ2, 527
    ES1, 596
    ES2, 598
    ESTOOLS2, 589
    EXPR2, 591
    FAMR2, 664
    FDIV2, 666
    FFCAT2, 784
    FLAGG2, 726
    FR2, 642
    FRAC2, 771
    FRIDEAL2, 753
    FRNAAF2, 773
    FS2, 799
    FSAGG2, 733
    IR2, 1230
    ITFUN2, 1093
    ITFUN3, 1095
    LIST2, 1437
    LIST3, 1440
    MATCAT2, 1515
    MMAP, 1600
    MPC2, 1582
    MPC3, 1584
    MRF2, 1568
    NSUP2, 2688
    OCTCT2, 2788
    ONECOMP2, 2799
    ORDCOMP2, 2808
    PARPC2, 2876
    PARSC2, 2878
    PARSU2, 2879
    PATRES2, 2930
    PATTERN2, 2895
    POLY2, 3020
    POLYLIFT, 2995
    PRIMARR2, 3092
    PTFUNC2, 2966
    QFCAT2, 3165
    QATCT2, 3168
    RMCAT2, 3287
    SEG2, 3372
    SEGBIND2, 3370
    STREAM2, 3429
    STREAM3, 3432
    SUP2, 3389
    ULS2, 3660
    UNISEG2, 3694
    UP2, 3672
    UPOLY2, 3662
    UPXS2, 3681
    UTS2, 3688
VECTOR2, 3718
mapBivariate
IBPTOOLS, 1227
mapdiv
STTAYLOR, 3437
mapExponents
PFORP, 2838
MAPPACK1, 1470
iter, 1470
recur, 1470
MAPPACK2, 1472
arg1, 1472
arg2, 1472
MAPPACK3, 1474
comp, 1474
mapMatrixIfCan
IBPTOOLS, 1227
mapmult
STTAYLOR, 3437
MappingPackage1, 1484
MappingPackage2, 1494
MappingPackage3, 1505
MappingPackage4, 1512
MappingPackageInternalHacks1, 1470
MappingPackageInternalHacks2, 1472
MappingPackageInternalHacks3, 1474
MAPPKG1, 1484
?**?, 1484
coece, 1484
fixedPoint, 1484
id, 1484
nullary, 1484
recur, 1484
MAPPKG2, 1494
const, 1495
constant, 1495
curry, 1495
diag, 1495
MAPPKG3, 1505
?*, 1505
constantLeft, 1505
constantRight, 1505
curryLeft, 1505
curryRight, 1505
twist, 1505
MAPPKG4, 1512
?**, 1512
?+?, 1512
?-?, 1512
?/?, 1512
mapSolve
SOLVEFOR, 3065
mapUnivariate
IBPTOOLS, 1227
mapUnivariateIfCan
IBPTOOLS, 1227
mat
ESTOOLS, 578
MATCAT2, 1515
map, 1515
reduce, 1515
match
LIST2MAP, 1442
mathieu11
PGE, 2945
mathieu12
PGE, 2945
mathieu22
PGE, 2945
mathieu23
PGE, 2945
mathieu24
PGE, 2945
MATLIN, 1520
adjoint, 1520
determinant, 1520
eColumn2, 1520
eRow1, 1520
eRow2, 1520
fractionFreeGauss, 1520
inverse, 1520
invertIfCan, 1520
minordet, 1520
normalizedDivide, 1520
nullity, 1520
nullSpace, 1520
rank, 1520
rowEchelon, 1520
MatrixCategoryFunctions2, 1515
MatrixCommonDenominator, 1517
matrixGcd
IBATOOL, 1223
<table>
<thead>
<tr>
<th>Function/Operation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixLinearAlgebraFunctions</td>
<td>1520</td>
</tr>
<tr>
<td>MatrixManipulation</td>
<td>1542</td>
</tr>
<tr>
<td>MATSTOR</td>
<td>3422</td>
</tr>
<tr>
<td>???</td>
<td>3422</td>
</tr>
<tr>
<td>copy</td>
<td>3422</td>
</tr>
<tr>
<td>leftScalarTimes</td>
<td>3422</td>
</tr>
<tr>
<td>minus</td>
<td>3422</td>
</tr>
<tr>
<td>plus</td>
<td>3422</td>
</tr>
<tr>
<td>power</td>
<td>3422</td>
</tr>
<tr>
<td>rightScalarTimes</td>
<td>3422</td>
</tr>
<tr>
<td>times</td>
<td>3422</td>
</tr>
<tr>
<td>maxPoints</td>
<td>948</td>
</tr>
<tr>
<td>maxrank</td>
<td>2862</td>
</tr>
<tr>
<td>maxrow</td>
<td>3512</td>
</tr>
<tr>
<td>MCALCFN</td>
<td>1602</td>
</tr>
<tr>
<td>bandedHessian</td>
<td>1602</td>
</tr>
<tr>
<td>bandedJacobian</td>
<td>1602</td>
</tr>
<tr>
<td>divergence</td>
<td>1602</td>
</tr>
<tr>
<td>gradient</td>
<td>1602</td>
</tr>
<tr>
<td>hessian</td>
<td>1602</td>
</tr>
<tr>
<td>jacobian</td>
<td>1602</td>
</tr>
<tr>
<td>laplacian</td>
<td>1602</td>
</tr>
<tr>
<td>MCDEN</td>
<td>1517</td>
</tr>
<tr>
<td>clearDenominator</td>
<td>1517</td>
</tr>
<tr>
<td>commonDenominator</td>
<td>1517</td>
</tr>
<tr>
<td>splitDenominator</td>
<td>1517</td>
</tr>
<tr>
<td>MDDFACT</td>
<td>1556</td>
</tr>
<tr>
<td>ddFact</td>
<td>1556</td>
</tr>
<tr>
<td>exptMod</td>
<td>1556</td>
</tr>
<tr>
<td>factor</td>
<td>1556</td>
</tr>
<tr>
<td>gcd</td>
<td>1556</td>
</tr>
<tr>
<td>linears</td>
<td>1556</td>
</tr>
<tr>
<td>separateFactors</td>
<td>1556</td>
</tr>
<tr>
<td>measure</td>
<td>56</td>
</tr>
<tr>
<td>INTPACK</td>
<td>56</td>
</tr>
<tr>
<td>ODEPACK</td>
<td>77</td>
</tr>
<tr>
<td>OPTPACK</td>
<td>67</td>
</tr>
<tr>
<td>PDEPACK</td>
<td>86</td>
</tr>
<tr>
<td>measure2Result</td>
<td>578</td>
</tr>
<tr>
<td>ESTOOLS</td>
<td>578</td>
</tr>
<tr>
<td>meatAxe</td>
<td></td>
</tr>
<tr>
<td>REP2</td>
<td>3338</td>
</tr>
<tr>
<td>mergeDifference</td>
<td></td>
</tr>
<tr>
<td>MTHING</td>
<td>1550</td>
</tr>
<tr>
<td>mergeFactors</td>
<td>644</td>
</tr>
<tr>
<td>GRDEF</td>
<td>948</td>
</tr>
<tr>
<td>PLEQN</td>
<td>2862</td>
</tr>
<tr>
<td>MESH</td>
<td>1552</td>
</tr>
<tr>
<td>meshFun2Var</td>
<td>1552</td>
</tr>
<tr>
<td>meshPar1Var</td>
<td>1552</td>
</tr>
<tr>
<td>meshPar2Var</td>
<td>1552</td>
</tr>
<tr>
<td>ptFunc</td>
<td>1552</td>
</tr>
<tr>
<td>MeshCreationRoutinesForThreeDimensions</td>
<td>1552</td>
</tr>
<tr>
<td>midpoint</td>
<td>3277</td>
</tr>
<tr>
<td>REAL0</td>
<td>3277</td>
</tr>
<tr>
<td>midpoints</td>
<td>966</td>
</tr>
<tr>
<td>GBINTERN</td>
<td>966</td>
</tr>
<tr>
<td>minimalForm</td>
<td>1130</td>
</tr>
<tr>
<td>PFOREP</td>
<td>2838</td>
</tr>
<tr>
<td>PLPKCRV</td>
<td>2984</td>
</tr>
<tr>
<td>minimalPolynomial</td>
<td></td>
</tr>
<tr>
<td>INBFF</td>
<td></td>
</tr>
<tr>
<td>minordet</td>
<td>1520</td>
</tr>
<tr>
<td>MATLIN</td>
<td>1520</td>
</tr>
<tr>
<td>minPoints</td>
<td>948</td>
</tr>
<tr>
<td>GRDEF</td>
<td>948</td>
</tr>
<tr>
<td>minPol</td>
<td>1420</td>
</tr>
<tr>
<td>LGROBP</td>
<td>1420</td>
</tr>
<tr>
<td>minPoly</td>
<td>14</td>
</tr>
<tr>
<td>AF</td>
<td>14</td>
</tr>
<tr>
<td>minrank</td>
<td>2862</td>
</tr>
<tr>
<td>PLEQN</td>
<td>2862</td>
</tr>
<tr>
<td>minset</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

PLEQN, 2862
minusInfinity
INFINITY, 1097
mix
PFOTOOLS, 2981
MKBCFUNC, 1454
binaryFunction, 1454
compiledFunction, 1454
MKFLCFN, 1456
MKFUNC, 1464
function, 1464
mkIntegral
CHVAR, 180
mkPrim
INTOOLS, 1242
MKRECORD, 1466
makeRecord, 1466
MKUCFUNC, 1468
compiledFunction, 1468
unaryFunction, 1468
MMAP, 1600
map, 1600
modTree
CRAPACK, 260
ModularDistinctDegreeFactorizer, 1556
modularFactor
GALFACT, 853
modularGcd
INMODGCD, 1113
modularGcdPrimitive
INMODGCD, 1113
ModularHermitianRowReduction, 1562
moduleSum
IBATOOL, 1223
moebiusMu
IN THEORY, 1197
monic polynomial, 3025
monic?
GALPOLYU, 870
monicCompleteDecompose
UPDECOMP, 3666
monicDecomposeIfCan
UPDECOMP, 3666
monicLeftDivide
OREPCTO, 3684
monicRightDivide
MonoidRingFunctions2, 1568
MonomialExtensionTools, 1570
monomialIntegrate
INTTR, 3567
monomialIntPoly
INTTR, 3567
monomials
PFORP, 2838
monicRDE
RDETR, 3592
monicRDEsys
RDETRS, 3597
MONOTOOL, 1570
decompose, 1570
normalDenom, 1570
split, 1570
splitSquarefree, 1570
more?
UDPO, 3696
moreAlgebraic?
QCMPACK, 3156
SFQCMPK, 3394
MoreSystemCommands, 1573
MPC2, 1582
map, 1582
reshape, 1582
MPC3, 1584
map, 1584
MPCPF, 1575
factor, 1575
MPolyCatFunctions2, 1582
MPolyCatFunctions3, 1584
MPolyCatPolyFactorizer, 1575
MPolyCatRationalFunctionFactorizer, 1578
MPRFF, 1578
factor, 1578
pushdown, 1578
pushdterm, 1578
pushlcoeff, 1578
pushlconst, 1578
pushup, 1578
INDEX

3887

totalfract, 1578
mpode
UTSODE, 3690
mr
TABLBUMP, 3512
MRATFAC, 1586
factor, 1586
MRationalFactorize, 1586
MRF2, 1568
map, 1568
MSYSCMD, 1573
systemCommand, 1573
MTHING, 1550
mergeDifference, 1550
mul
POLYVEC, 3704
mulbybinomial
POLYVEC, 3704
mulbyscalar
POLYVEC, 3704
MULTFACT, 1607
factor, 1607
MultFiniteFactorize, 1589
multiEuclideanTree
CRAFACK, 260
multinomial
COMBINAT, 1167
multiple
PMASS, 2900
PMASSFS, 790
MultipleMap, 1600
multiplicity
NPOLYGON, 2692
PLPKCRV, 2984
multisect
STTAYLOR, 3437
multiServ
AXSERV, 115
MultiVariableCalculusFunctions, 1602
multivariate
INTG0, 932
POLYCATQ, 2997
RF, 3212
MultivariateFactorize, 1607
MultivariateSquareFree, 1615
MULTSQFR, 1615
check, 1615
ccoefChoose, 1615
compdegd, 1615
consnewpol, 1615
intChoose, 1615
lift, 1615
myDegree, 1615
normDeriv2, 1615
nsqfree, 1615
squareFree, 1615
squareFreePrim, 1615
univcase, 1615
musserTrials
GALFACT, 853
myDegree
MULTSQFR, 1615
NAGC02, 2451
c02aff, 2451
c02agf, 2451
NAGC05, 2470
c05adg, 2470
c05mgb, 2470
c05pbg, 2470
NAGC06, 2518
c06mab, 2518
c06mef, 2518
c06ef, 2518
c06f, 2518
c06fpe, 2518
c06fuf, 2518
c06gbf, 2518
c06gcf, 2518
c06gqf, 2518
c06gsf, 2518
NAGD01, 1999
d01ajf, 1999
d01akf, 1999
d01afl, 1999
d01amf, 1999
d01amf, 1999
d01apf, 1999
d01aqf, 1999
d01asf, 1999
INDEX

3888

d01bbf, 1999
d01cfc, 1999
d01gaf, 1999
d01gbf, 1999

NAGD02, 2386

d02bbf, 2386
d02bhf, 2386
d02cjf, 2386
d02ejf, 2386
d02gaf, 2386
d02gbf, 2386
d02kef, 2386
d02raf, 2386

NAGD03, 2433

d03edf, 2433
d03eef, 2433
d03raf, 2433

NAGE01, 2047

e01baf, 2047
e01bef, 2047
e01bff, 2047
e01bgf, 2047
e01bhf, 2047
e01daf, 2047
e01saf, 2047
e01sbf, 2047
e01sef, 2047
e01sff, 2047

NAGE02, 1832

e02adf, 1832
e02aef, 1832
e02agf, 1832
e02ahf, 1832
e02ajf, 1832
e02akf, 1832
e02baf, 1832
e02bbf, 1832
e02bcf, 1832
e02ddf, 1832

NAGE04, 2287

e04dgf, 2287
e04fdf, 2287
e04gcf, 2287
e04jaf, 2287
e04mbf, 2287
e04naf, 2287
e04ucf, 2287
e04ycf, 2287

NagEigenPackage, 1690

NAGF01, 2128

f01brf, 2128
f01bsf, 2128
f01maf, 2128
f01mcf, 2128
f01qcf, 2128
f01qdf, 2128
f01qef, 2128
f01rcf, 2128
f01rdrf, 2128
f01reff, 2128

NAGF02, 1690

f02aaf, 1690
f02abf, 1690
f02adf, 1690
f02af, 1690
f02aff, 1690
f02agf, 1690
f02ajf, 1690
f02akf, 1690
f02awf, 1690
f02axf, 1690
f02axf, 1690
f02bjf, 1690
f02fjf, 1690
f02jff, 1690
f02kff, 1690
f02wef, 1690
f02xef, 1690

NAGF04, 1910

f04adg, 1911
f04arf, 1911
f04asf, 1911
f04af, 1911
f04axf, 1911
f04jgf, 1911
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f04maf</td>
<td>1911</td>
</tr>
<tr>
<td>f04mbf</td>
<td>1911</td>
</tr>
<tr>
<td>f04mcf</td>
<td>1911</td>
</tr>
<tr>
<td>f04qaf</td>
<td>1911</td>
</tr>
<tr>
<td>NAGF07</td>
<td>2068</td>
</tr>
<tr>
<td>f07adf</td>
<td>2068</td>
</tr>
<tr>
<td>f07aef</td>
<td>2068</td>
</tr>
<tr>
<td>f07ddf</td>
<td>2068</td>
</tr>
<tr>
<td>f07def</td>
<td>2068</td>
</tr>
<tr>
<td>NagFittingPackage</td>
<td>1832</td>
</tr>
<tr>
<td>NagIntegrationPackage</td>
<td>1999</td>
</tr>
<tr>
<td>NagInterpolationPackage</td>
<td>2047</td>
</tr>
<tr>
<td>NagLapack</td>
<td>2068</td>
</tr>
<tr>
<td>NagLinearEquationSolvingPackage</td>
<td>1910</td>
</tr>
<tr>
<td>NAGLinkSupportPackage</td>
<td>1920</td>
</tr>
<tr>
<td>NagMatrixOperationsPackage</td>
<td>2128</td>
</tr>
<tr>
<td>NagOptimisationPackage</td>
<td>2287</td>
</tr>
<tr>
<td>NagOrdinaryDifferentialEquationsPackage</td>
<td>2386</td>
</tr>
<tr>
<td>NagPartialDifferentialEquationsPackage</td>
<td>2433</td>
</tr>
<tr>
<td>NagPolynomialRootsPackage</td>
<td>2451</td>
</tr>
<tr>
<td>NagRootFindingPackage</td>
<td>2470</td>
</tr>
<tr>
<td>NAGS</td>
<td>2671</td>
</tr>
<tr>
<td>s01eaf</td>
<td>2671</td>
</tr>
<tr>
<td>s13aaf</td>
<td>2671</td>
</tr>
<tr>
<td>s13acf</td>
<td>2671</td>
</tr>
<tr>
<td>s13adf</td>
<td>2671</td>
</tr>
<tr>
<td>s14aaf</td>
<td>2671</td>
</tr>
<tr>
<td>s14abf</td>
<td>2671</td>
</tr>
<tr>
<td>s14baf</td>
<td>2671</td>
</tr>
<tr>
<td>s15adf</td>
<td>2671</td>
</tr>
<tr>
<td>s15aef</td>
<td>2671</td>
</tr>
<tr>
<td>s17aef</td>
<td>2671</td>
</tr>
<tr>
<td>s17aef</td>
<td>2671</td>
</tr>
<tr>
<td>s17aff</td>
<td>2671</td>
</tr>
<tr>
<td>s17aff</td>
<td>2671</td>
</tr>
<tr>
<td>s17agf</td>
<td>2671</td>
</tr>
<tr>
<td>s17agf</td>
<td>2671</td>
</tr>
<tr>
<td>s17ahf</td>
<td>2671</td>
</tr>
<tr>
<td>s17ajf</td>
<td>2671</td>
</tr>
<tr>
<td>s17akf</td>
<td>2671</td>
</tr>
<tr>
<td>s17dcf</td>
<td>2671</td>
</tr>
<tr>
<td>s17def</td>
<td>2671</td>
</tr>
<tr>
<td>s17dgf</td>
<td>2671</td>
</tr>
<tr>
<td>s17dhf</td>
<td>2671</td>
</tr>
<tr>
<td>s18acf</td>
<td>2671</td>
</tr>
<tr>
<td>s18adff</td>
<td>2671</td>
</tr>
<tr>
<td>s18aff</td>
<td>2671</td>
</tr>
<tr>
<td>s18def</td>
<td>2671</td>
</tr>
<tr>
<td>s18dcf</td>
<td>2671</td>
</tr>
<tr>
<td>s19aaf</td>
<td>2671</td>
</tr>
<tr>
<td>s19abf</td>
<td>2671</td>
</tr>
<tr>
<td>s19acf</td>
<td>2671</td>
</tr>
<tr>
<td>s19adff</td>
<td>2671</td>
</tr>
<tr>
<td>s20aaf</td>
<td>2671</td>
</tr>
<tr>
<td>s20abf</td>
<td>2671</td>
</tr>
<tr>
<td>s21baf</td>
<td>2671</td>
</tr>
<tr>
<td>s21bbf</td>
<td>2671</td>
</tr>
<tr>
<td>s21bcf</td>
<td>2671</td>
</tr>
<tr>
<td>s21bdf</td>
<td>2671</td>
</tr>
<tr>
<td>NagSeriesSummationPackage</td>
<td>2518</td>
</tr>
<tr>
<td>NAGSP</td>
<td>1920</td>
</tr>
<tr>
<td>aspFilename</td>
<td>1920</td>
</tr>
<tr>
<td>checkPrecision</td>
<td>1920</td>
</tr>
<tr>
<td>dimensionsOf</td>
<td>1920</td>
</tr>
<tr>
<td>fortranCompilerName</td>
<td>1920</td>
</tr>
<tr>
<td>fortranLinkerArgs</td>
<td>1920</td>
</tr>
<tr>
<td>restorePrecision</td>
<td>1920</td>
</tr>
<tr>
<td>NagSpecialFunctionsPackage</td>
<td>2671</td>
</tr>
<tr>
<td>NCEP</td>
<td>2774</td>
</tr>
<tr>
<td>characteristicPolynomial</td>
<td>2774</td>
</tr>
<tr>
<td>complexEigenvalues</td>
<td>2774</td>
</tr>
<tr>
<td>complexEigenvectors</td>
<td>2774</td>
</tr>
<tr>
<td>continuedFraction</td>
<td>2777</td>
</tr>
<tr>
<td>NCODIV</td>
<td>2698</td>
</tr>
<tr>
<td>leftDivide</td>
<td>2698</td>
</tr>
<tr>
<td>leftExactQuotient</td>
<td>2698</td>
</tr>
<tr>
<td>leftGcd</td>
<td>2698</td>
</tr>
<tr>
<td>leftLcm</td>
<td>2698</td>
</tr>
<tr>
<td>leftQuotient</td>
<td>2698</td>
</tr>
<tr>
<td>leftRemainder</td>
<td>2698</td>
</tr>
<tr>
<td>negAndPosEdge</td>
<td></td>
</tr>
<tr>
<td>NPOLYGON</td>
<td>2692</td>
</tr>
<tr>
<td>neglist</td>
<td></td>
</tr>
<tr>
<td>ESTOOLS1</td>
<td>587</td>
</tr>
<tr>
<td>newLine</td>
<td></td>
</tr>
<tr>
<td>DISPLAY</td>
<td>324</td>
</tr>
<tr>
<td>newReduc</td>
<td></td>
</tr>
<tr>
<td>FSRED</td>
<td>809</td>
</tr>
<tr>
<td>NewSparseUnivariatePolynomialFunctions2</td>
<td>2688</td>
</tr>
<tr>
<td>NEWTON</td>
<td>2690</td>
</tr>
</tbody>
</table>
newton, 2690
NewtonInterpolation, 2690
NewtonPolygon, 2692
NewtonPolySlope
BLUPPACK, 164
nextColeman
SGCF, 3498
nextIrreduciblePoly
FFPOLY, 695
nextLatticePermutation
SGCF, 3498
nextNormalPoly
FFPOLY, 695
nextNormalPrimitivePoly
FFPOLY, 695
nextPartition
SGCF, 3498
nextPrime
PRIMES, 1203
nextPrimitiveNormalPoly
FFPOLY, 695
nextPrimitivePoly
FFPOLY, 695
nextsousResultant2
PRS, 3117
nextSublist
PLEQN, 2862
nextSubsetGray
GRAY, 951
NFINTBAS, 2726
discriminant, 2726
integralBasis, 2726
localIntegralBasis, 2726
nlde
STTAYLOR, 3437
NLINSOL, 2708
solve, 2708
solveInField, 2708
NODE1, 2703
solve, 2703
noKaratsuba
UPMP, 3674
noLinearFactor?
BRILL, 174
NonCommutativeOperatorDivision, 2698
NONE1, 2701
coerce, 2701
NonFunctions1, 2701
NonLinearFirstOrderODESolver, 2703
nonLinearPart
E04AGNT, 629
NonLinearSolvePackage, 2708
nonQsign
TOOLSIGN, 3531
norm
CRFP, 233
GALFACTU, 848
INBFF, 1130
NORMMA, 2716
normal
RFDIST, 3201
normal form, 3025
normal01
RFDIST, 3201
normal?
FFPOLY, 695
INBFF, 1130
normalDenom
MONOTOOL, 1570
normalDeriv
FACUTIL, 647
normalElement
INBFF, 1130
normalForm
GB, 1002
PSEUDLIN, 3112
normalise
REP, 3172
NormalizationPackage, 2711
normalize
EFSTRUC, 493
NORMPK, 2711
normalizedAssociate
NORMPK, 2711
normalizedDivide
MATLIN, 1520
MHROWRED, 1562
normalizeIfCan
INDEX

LAZM3PK, 1313
normDeriv2
MULTSQFR, 1615
normFactors
NORMRETR, 2719
NormInMonogenicAlgebra, 2716
normInvertible?
NORMPK, 2711
NORMMA, 2716
norm, 2716
NORMPK, 2711
normalize, 2711
normalizedAssociate, 2711
normInvertible?, 2711
outputArgs, 2711
recip, 2711
NORMRETR, 2719
Frobenius, 2719
normFactors, 2719
retractIfCan, 2719
NormRetractPackage, 2719
NP COEF, 2721
listexp, 2721
npcoef, 2721
NP Coef, 2721
npcoef
NP COEF, 2721
N POLYGON, 2692
multiplicity, 2692
negAndPosEdge, 2692
newtonPolygon, 2692
slope, 2692
NREP, 2779
characteristicPolynomial, 2780
realEigenvalues, 2780
realEigenvectors, 2780
nsqfree
MULTSQFR, 1615
NSUP2, 2688
map, 2688
nthr
POLYROOT, 3042
nthRoot
FACTFUNC, 638
N TPOLFN, 2737
bernoulliB, 2737
cyclotomic, 2737
eulerE, 2737
nullary
MAPPKG1, 1484
nullity
MATLIN, 1520
nullSpace
IMATQF, 1111
MATLIN, 1520
NumberFieldIntegralBasis, 2726
NumberFormats, 2732
numberOfDivisors
INTHEORY, 1197
numberOfFactors
GALFACT, 853
numberOfImproperPartitions
SGCF, 3498
numberOfIrreduciblePoly
FFPOLY, 695
numberOfNormalPoly
FFPOLY, 695
numberOfOperations
ESTOOLS, 578
numberOfPlacesOfDegree
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
numberOfPrimitivePoly
FFPOLY, 695
numberOfValuesNeeded
RECOP, 3289
numberOfVariables
RSDCMPK, 3310
SRDCMPK, 3404
numberPlacesDegExtDeg
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
numberRatPlacesExtDeg
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
NumberTheoreticPolynomialFunctions, 2737
NUMERIC, 2740
complexNumeric, 2740
complexNumericIfCan, 2740
numeric, 2740
numericICan, 2740
Numeric, 2740
numeric
NUMERIC, 2740
NumericalOrdinaryDifferentialEquations, 2752
NumericalQuadrature, 2762
NumericComplexEigenPackage, 2774
NumericContinuedFraction, 2777
numericICan
NUMERIC, 2740
NumericRealEigenPackage, 2779
NumericTubePlot, 2782
NUMFMT, 2732
FormatArabic, 2732
FormatRoman, 2732
ScanArabic, 2732
ScanFloatIgnoreSpaces, 2732
ScanFloatIgnoreSpacesICan, 2732
ScanRoman, 2732
NUMODE, 2752
rk4, 2752
rk4a, 2752
rk4f, 2752
rk4qc, 2752
NUMQUAD, 2762
aromberg, 2762
asimpson, 2762
atrapezoidal, 2762
romberg, 2762
rombergo, 2762
simpson, 2762
simpsono, 2762
trapezoidal, 2762
trapezoidalo, 2762
NUMTUBE, 2782
tube, 2782
oblatespheroidal
COORDSYS, 255
ocf2ocdf
ESTOOLS, 578
OCTCT2, 2788
map, 2788
OctonionCategoryFunctions2, 2788
oddInfiniteProduct
INFPRODO, 1085
INFPRODF, 1087
INFPRODPF, 1091
STINPROD, 3434
odddatons
STTAYLOR, 3437
oddlambert
STTAYLOR, 3437
ode
UTSODE, 3690
ode1
UTSODE, 3690
ode2
UTSODE, 3690
ODECONST, 252
constDsolve, 252
ODEEF, 481
solve, 481
ODEINT, 2790
diff, 2790
expint, 2790
int, 2790
ODEIntegration, 2790
ODEPACK, 77
measure, 77
solve, 77
ODEPAL, 3146
algDsolve, 3146
ODEPRIM, 3098
denomLODE, 3098
indicialEquation, 3098
indicialEquations, 3098
splitDenominator, 3098
ODEPRRIC, 3103
changeVar, 3103
constantCoefficientRicDE, 3103
denomRicDE, 3103
leadingCoefficientRicDE, 3103
polyRicDE, 3103
singRicDE, 3103
ODERAT, 3248
indicialEquationAtInfinity, 3248
ratDsolve, 3248
ODERED, 3304
reduceLODE, 3304
ODERTRIC, 3256
polyRicDE, 3256
ricDsolve, 3256
singRicDE, 3256
ODESYS, 3485
solve, 3485
solveInField, 3485
triangulate, 3485
ODETOOLS, 2793
particularSolution, 2793
variationOfParameters, 2793
wronskianMatrix, 2793
ODETools, 2793
OMEXPR, 606
OMwrite, 606
OMlistCDs
OMPKG, 2801
OMlistSymbols
OMPKG, 2801
OMPKG, 2801
OMlistCDs, 2801
OMlistSymbols, 2801
OMread, 2801
OMreadFile, 2801
OMreadStr, 2801
OMsupportsCD?, 2801
OMsupportsSymbol?, 2801
OMunhandledSymbol, 2801
OMread
OMPKG, 2801
OMreadFile
OMPKG, 2801
OMreadStr
OMPKG, 2801
OMreceive
OMSERVER, 2804
OMsend
OMSERVER, 2804
OMserve
OMSERVER, 2804
OMSERVER, 2804
OMreceive, 2804
OMsend, 2804
OMserve, 2804
OMsupportsCD?
OMPKG, 2801
OMsupportsSymbol?
3894

totalLex, 2810
OREPCTO, 3684
  apply, 3684
  leftDivide, 3684
  monicLeftDivide, 3684
  monicRightDivide, 3684
  rightDivide, 3684
  times, 3684
OrthogonalPolynomialFunctions, 2813
  orthonormalBasis
    REP, 3172
  ORTHPOL, 2813
    chebyshevT, 2813
    chebyshevU, 2813
    hermiteH, 2813
    laguerreL, 2813
    legendreP, 2813
  OUT, 2816
    output, 2816
    outputList, 2816
  output
    OUT, 2816
  outputArgs
    NORMPK, 2711
  outputAsFortran
    FORT, 750
    SPECOUT, 3391
  outputAsScript
    SPECOUT, 3391
  outputAsTex
    SPECOUT, 3391
  outputList
    OUT, 2816
  outputMeasure
    ESTOOLS, 578
  OutputPackage, 2816
  outset?
    PLEQN, 2862

packageCall
  INFORM1, 1159
PackageForAlgebraicFunctionField, 2821
PackageForAlgebraicFunctionFieldOverFiniteField, 2829
PackageForPoly, 2838
PADE, 2848
  pade, 2848
  pade, 2848
  pade, 2848
  pade, 2848
  PADE, 2848
  PADEPAC, 2846
  PadeApproximantPackage, 2846
  PadeApproximants, 2848
  pade
  PADE, 2848
  PADEPAC, 2846
  pade, 2846
PAAdicWildFunctionFieldIntegralBasis, 2853
PAFF, 2821
  adjunctionDivisor, 2821
  classNumber, 2821
  desingTree, 2821
  desingTreeWoFullParam, 2821
  eval, 2821
  evalIFcan, 2821
  findOrderOfDivisor, 2821
  fullDesTree, 2821
  fullInfClsPt, 2821
  genus, 2821
  genusNeg, 2821
  goppaCode, 2821
  homogenize, 2821
  interpolateForms, 2821
  interpolateFormsForFact, 2821
  intersectionDivisor, 2821
  lBasis, 2821
  LPolynomial, 2821
  numberOfPlacesOfDegree, 2821
  numberPlacesDegExtDeg, 2821
  numberRatPlacesExtDeg, 2821
  parametrize, 2821
  placesAbove, 2821
  placesOfDegree, 2821
  pointDominateBy, 2821
  projectivePoint, 2821
  rationalPlaces, 2821
  rationalPoints, 2821
  setCurve, 2821
  setSingularPoints, 2821
  singularPoints, 2821
  theCurve, 2821
  ZetaFunction, 2821
INDEX

PAFFFF, 2829
  adjunctionDivisor, 2829
  classNumber, 2829
  desingTree, 2829
  desingTreeWoFullParam, 2829
  eval, 2829
  evalIfCan, 2829
  findOrderOfDivisor, 2829
  fullDesTree, 2829
  fullInfClsPt, 2829
  genus, 2829
  genusNeg, 2829
  goppaCode, 2829
  homogenize, 2829
  interpolateForms, 2829
  interpolateFormsForFact, 2829
  intersectionDivisor, 2829
  lBasis, 2829
  LPolynomial, 2829
  numberOfPlacesOfDegree, 2829
  numberPlacesDegExtDeg, 2829
  numberRatPlacesExtDeg, 2829
  parametrize, 2829
  placesAbove, 2829
  placesOfDegree, 2829
  pointDominateBy, 2829
  projectivePoint, 2829
  rationalPlaces, 2829
  rationalPoints, 2829
  setCurve, 2829
  setSingularPoints, 2829
  singularPoints, 2829
  theCurve, 2829
  translateToOrigin, 2829
  ZetaFunction, 2829

palgextint
  INTPAF, 3137
palgextint0
  INTG0, 932

palgfldint
  INTALG, 23

palglint
  palglint
    INTPAF, 3137
    INTG0, 932
palglint0
  INTG0, 932

palgintegrate
  PAN2EXPR, 2993
  coerce, 2993

parabolic
  COORDSYS, 255
  parabolicCylindrical
    COORDSYS, 255
paraboloidal
  COORDSYS, 255
ParadoxicalCombinatorsForStreams, 2859
ParametricLinearEquations, 2862
ParametricPlaneCurveFunctions2, 2876
ParametricSpaceCurveFunctions2, 2878
ParametricSurfaceFunctions2, 2879
ParametrizationPackage, 2881
parametrize
  GPAFF, 889
  PAFF, 2821
  PAFFF, 2829
  PARAMP, 2881
PARAMP, 2881
  parametrize, 2881
ParCond
  PLEQN, 2862
ParCondList
  PLEQN, 2862
PARPC2, 2876
  map, 2876
PARSC2, 2878
  map, 2878
PARSU2, 2879
  map, 2879
partialFraction
  PFRPAC, 2886
PartialFractionPackage, 2886
particularSolution
  LSMP, 1412
  LSMP1, 1415
  ODETOOLS, 2793
  SOLVEFOR, 3065
partition
  COMBINAT, 1167
partitions
  PARTPERM, 2888
PartitionsAndPermutations, 2888
PARTPERM, 2888
  conjugate, 2888
  conjugates, 2888
  partitions, 2888
  permutations, 2888
  sequences, 2888
  shuffle, 2888
  shufflein, 2888
pascalTriangle
  GALUTIL, 873
PATMATCH, 2897
  Is, 2897
  is?, 2897
PATRES2, 2930
  map, 2930
PATTERN1, 2892
  addBadValue, 2892
  badValues, 2892
  predicate, 2892
  satisfy?, 2892
  suchThat, 2892
PATTERN2, 2895
  map, 2895
PatternFunctions1, 2892
PatternFunctions2, 2895
PatternMatch, 2897
patternMatch
  CPMATCH, 230
  PMDOWN, 2925
  PMFS, 2902
  PMNS, 2905
  PMKERNEL, 2916
  PMLSAGG, 2919
  PMPLCAT, 2922
  PMQFCAT, 2928
  PMSYM, 2932
  PMTOOLS, 2934
PatternMatchAssertions, 2900
PatternMatchFunctionSpace, 2902
PatternMatchIntegerNumberSystem, 2905
PatternMatchIntegration, 2908
PatternMatchKernel, 2916
PatternMatchListAggregate, 2919
PatternMatchPolynomialCategory, 2922
PatternMatchPushDown, 2925
PatternMatchQuotientFieldCategory, 2928
PatternMatchResultFunctions2, 2930
PatternMatchSymbol, 2932
patternMatchTimes
  PMTOOLS, 2934
PatternMatchTools, 2934
PDECOMP, 3003
  decompose, 3003
  leftFactor, 3003
  rightFactorCandidate, 3003
PDEPACK, 86
  measure, 86
  solve, 86
pdf2df
  ESTOOLS, 578
pdf2ef
  ESTOOLS, 578
perfectNthPower?
  IROOT, 1216
perfectNthRoot
  IROOT, 1216
perfectSqrt
  IROOT, 1216
perfectSquare?
  IROOT, 1216
PERMAN, 2940
  permanent, 2940
Permanent, 2940
permanent
  PERMAN, 2940
permutation
  COMBF, 198
  COMBINAT, 1167
PermutationGroupExamples, 2945
INDEX

permutationRepresentation
REP1, 3330

permutations
PARTPERM, 2888

PFBR, 3006
bivariateSLPEBR, 3006
factorByRecursion, 3006
factorSFBRelUnit, 3006
factorSquareFreeByRecursion, 3006
randomR, 3006
solveLinearPolynomialEquationByRecursion, 3006

PFBRU, 3013
factorByRecursion, 3013
factorSFBRelUnit, 3013
factorSquareFreeByRecursion, 3013
randomR, 3013
solveLinearPolynomialEquationByRecursion, 3013

PFO, 2971
order, 2971
torsion?, 2971
torsionIfCan, 2971

PFOQ, 2978
order, 2978
torsion?, 2978
torsionIfCan, 2978

PFORP, 2838
constant, 2838
degOneCoef, 2838
degree, 2838
degreeOfMinimalForm, 2838
firstExponent, 2838
homogenize, 2838
listAllMono, 2838
listAllMonoExp, 2838
listVariable, 2838
mapExponents, 2838
minimalForm, 2838
monomials, 2838
replaceVarByOne, 2838
replaceVarByZero, 2838
subs1stVar, 2838
subs2ndVar, 2838
subsInVar, 2838
totalDegree, 2838
translate, 2838
univariate, 2838

PFOTOOLS, 2981
badNum, 2981
doubleDisc, 2981
getGoodPrime, 2981
mix, 2981
polyred, 2981

PFRPAC, 2886
partialFraction, 2886

PGCD, 3022
gcd, 3022
gcdPrimitive, 3022

PGE, 2945
abelianGroup, 2945
alternatingGroup, 2945
cyclicGroup, 2945
dihedralGroup, 2945
janko2, 2945
mathieu11, 2945
mathieu12, 2945
mathieu22, 2945
mathieu23, 2945
mathieu24, 2945
rubiksGroup, 2945
symmetricGroup, 2945
youngGroup, 2945

PGROEB, 2990
lexGroebner, 2990
totalGroebner, 2990

phiCoord
PTPACK, 2968

pi

EF, 450

PICOERCE, 2954
coerce, 2954
PiCoercions, 2954

PINTERP, 3033
interpolate, 3033

PINTERPA, 3035
LagrangeInterpolation, 3035

placesAbove

GPAFF, 889
PAFF, 2821
PAFFFF, 2829

placesOfDegree
INDEX

GPAFF, 889
INTDIVP, 1261
PAFF, 2821
PAFFFF, 2829
PLEQN, 2862

B1solve, 2862
bsolve, 2862
dmp2rfi, 2862
factorset, 2862
hasln, 2862
inconsistent?, 2862
maxrank, 2862
minrank, 2862
minset, 2862
nextSublist, 2862
overset?, 2862
ParCond, 2862
ParCondList, 2862
pr2dmp, 2862
psolve, 2862
rdregime, 2862
redmat, 2862
redpsps, 2862
regime, 2862
se2rfi, 2862
sqfree, 2862
wrregime, 2862

pleskenSplit
CRFP, 233

plot
PLOT1, 2956
PLOT1, 2956
plot, 2956
plotPolar, 2956
PlotFunctions1, 2956
plotPolar
PLOT1, 2956
PLOTTOOL, 2958
calcRanges, 2958
PlotTools, 2958
PLPKCRV, 2984
eval, 2984
minimalForm, 2984
multiplicity, 2984
pointInIdeal?, 2984
translateToOrigin, 2984

plusInfinity
INFINITY, 1097
PMASS, 2900
assert, 2900
constant, 2900
multiple, 2900
optional, 2900
PMASSFS, 790
assert, 790
constant, 790
multiple, 790
optional, 790
pmComplexintegrate
INTPM, 2908
PMDOWN, 2925
fixPredicate, 2925
patternMatch, 2925
PMFS, 2902
patternMatch, 2902
PMINS, 2905
patternMatch, 2905
pmintegrate
INTPM, 2908
PMKERNEL, 2916
patternMatch, 2916
PMLSAGG, 2919
patternMatch, 2919
PMPLCAT, 2922
patternMatch, 2922
PMPRED, 113
suchThat, 113
PMPREDFS, 793
suchThat, 793
PMQFCAT, 2928
patternMatch, 2928
PMSYM, 2932
patternMatch, 2932
PMTOOLS, 2934
patternMatch, 2934
patternMatchTimes, 2934
PNTHEORY, 3037
bernoulli, 3037
chebyshevT, 3037
chebyshevU, 3037
cyclotomic, 3037
euler, 3037
fixedDivisor, 3037
hermite, 3037
laguerre, 3037
legendre, 3037
point
"TUBETOOL, 3631
pointColorDefault
VIEWDEF, 3720
pointColorPalette
DROPT0, 399
pointDominatedBy
GPAFF, 889
LPARSPT, 1445
PAFF, 2821
PAFFFF, 2829
PointFunctions2, 2966
pointInIdeal?
"PLPKCRV, 2984
PointPackage, 2968
pointSizeDefault
VIEWDEF, 3720
PointsOfFiniteOrder, 2971
PointsOfFiniteOrderRational, 2978
PointsOfFiniteOrderTools, 2981
pointToPlace
LPARSPT, 1445
poisson
RIDIST, 3204
pol
INBFF, 1130
polar
"COORDSYS, 255
polCase
"LEADCDET, 1317
PoK
FFSQFR, 722
PollardSmallFactor
INTFACT, 1172
POLTOPOL, 2987
dmpToHdmp, 2987
dmpToP, 2987
hdmpToDmp, 2987
hdmpToP, 2987
pToDmp, 2987
pToHdmp, 2987
PolToPol, 2987
POLUTIL, 3268
boundOfCauchy, 3268
lazyVariations, 3268
sturmSequence, 3268
sturmVariationsOf, 3268
sylvesterSequence, 3268
POLY2, 3020
map, 3020
POLY2UP, 3075
univariate, 3075
POLYCATQ, 2997
isExpt, 2997
isPlus, 2997
isPower, 2997
isTimes, 2997
mainVariable, 2997
multivariate, 2997
univariate, 2997
variables, 2997
polygamma
"DFSFUN, 348
FSPECF, 775
PolyGroebner, 2990
POLYLIFT, 2995
map, 2995
PolynomialAN2Expression, 2993
PolynomialCategoryLifting, 2995
PolynomialCategoryQuotientFunctions, 2997
PolynomialComposition, 3001
PolynomialDecomposition, 3003
PolynomialFactorizationByRecursion, 3006
PolynomialFactorizationByRecursionUnivariate, 3013
PolynomialFunctions2, 3020
PolynomialGcdPackage, 3022
PolynomialInterpolation, 3033
PolynomialInterpolationAlgorithms, 3035
PolynomialNumberTheoryFunctions, 3037
PolynomialPackageForCurve, 2984
PolynomialRoots, 3042
PolynomialSetUtilitiesPackage, 3046
PolynomialSolveByFormulas, 3064
PolynomialSquareFree, 3071
PolynomialToUnivariatePolynomial, 3075
polynomialZeros
"ESCONT, 569
polyRDE
   RDETR, 3592
polyred
   PFOTOOLS, 2981
polyRicDE
   ODEPRRIC, 3103
   ODERTRIC, 3256
polyRing2UPUP
   AFALGRES, 9
polyRingToBlUpRing
   BLUPPACK, 164
POLYROOT, 3042
   froot, 3042
   nthr, 3042
   qroot, 3042
   rroot, 3042
POLYVEC, 3704
   copyfirst, 3704
   copyslice, 3704
   degree, 3704
   differentiate, 3704
   divide, 3704
   evalat, 3704
   extendedgcd, 3704
   gcd, 3704
   lcm, 3704
   mul, 3704
   mulbybinomial, 3704
   multyscalar, 3704
   pow, 3704
   remainder, 3704
   resultant, 3704
   tomodpa, 3704
   truncatedmuladd, 3704
   truncatedmultiplication, 3704
   vectoraddmul, 3704
   vectorcombination, 3704
popFortranOutputStack
   FOP, 747
positiveSolve
   ZDSOLVE, 3812
possiblyNewVariety?
   PSETPK, 3047
pow
   POLYVEC, 3704
   power
   STTAYLOR, 3437
power
   STTAYLOR, 3437
PowerSeriesLimitPackage, 3077
powerSum
   CYCLES, 281
pr2dmp
   PLEQN, 2862
PREASSOC, 3089
   firstUncouplingMatrix, 3089
PrecomputedAssociatedEquations, 3089
predicate
   PATTERN1, 2892
prepareDecompose
   QCMPPACK, 3156
   SFQCMPPK, 3394
prepareSubResAlgo
   RSETGCD, 3317
prevPrime
   PRIMES, 1203
PRIMARR2, 3092
   map, 3092
   reduce, 3092
   scan, 3092
primaryDecomp
   IDECOMP, 1074
prime?
   GAUSSFAC, 877
   IDECOMP, 1074
   PRIMES, 1203
PRIMELT, 3095
   primitiveElement, 3095
PRIMES, 1203
   nextPrime, 1203
   prevPrime, 1203
   prime?, 1203
   primes, 1203
primes
   PRIMES, 1203
primextendedint
   INTTR, 3567
primextendintfrac
   INTTR, 3567
primintegrate
   INTTR, 3567
primintfdpoly
INDEX 3901

INTTR, 3567
primitive?
FFPOLY, 695
PrimitiveArrayFunctions2, 3092
PrimitiveElement, 3095
primitiveElement
FSPRMELT, 805
PRIMELT, 3095
primitivePart
SUBRESP, 3479
PrimitiveRatDE, 3098
PrimitiveRatRicDE, 3103
primlimintfrac
INTTR, 3567
primlimitedint
INTTR, 3567
print
GBINTERN, 966
printINFO
GBINTERN, 966
primpolINFO
GBINTERN, 966
prinshINFO
GBINTERN, 966
PRINT, 3110
print, 3110
print
PRINT, 3110
printInfo
GPAFF, 889
LPARSPRT, 1445
RSDCMPK, 3310
SRDCMPK, 3404
printingInfo?
TBCMPPK, 3516
PrintPackage, 3110
PRJALGPK, 2961
algebraicSet, 2961
rationalPoints, 2961
singularPoints, 2961
singularPointsWithRestriction, 2961
probablyZeroDim?
PSETPK, 3047
problemPoints
D01AGNT, 406
ESCONT, 569
product
COMBF, 198
ProjectiveAlgebraicSetPackage, 2961
projectivePoint
PAFF, 2821
PAFFFF, 2829
prolateSpheroidal
COORDSYS, 255
PRS, 3116
?*, 3117
chainSubResultants, 3117
degreeSubResultant, 3117
degreeSubResultantEuclidean, 3117
discriminant, 3117
discriminantEuclidean, 3117
divide, 3117
gcd, 3117
exquo, 3117
indiceSubResultant, 3117
indiceSubResultantEuclidean, 3117
lastSubResultant, 3117
lastSubResultantEuclidean, 3117
Lazard, 3117
Lazard2, 3117
nextsousResultant2, 3117
pseudoDivide, 3117
resultant, 3117
resultantEuclidean, 3117
resultantEuclideannaif, 3117
resultantnaif, 3117
resultantReduit, 3117
resultantReduitEuclidean, 3117
schema, 3117
semiDegreeSubResultantEuclidean, 3117
semiDiscriminantEuclidean, 3117
semiIndiceSubResultantEuclidean, 3117
semiLastSubResultantEuclidean, 3117
semiResultantEuclideannaif, 3117
semiResultantEuclidean1, 3117
semiResultantEuclidean2, 3117
semiResultantEuclideannaif, 3117
semiResultantReduitEuclidean, 3117
semiSubResultantGcdEuclideannaif, 3117
semiSubResultantGcdEuclidean1, 3117
semiSubResultantGcdEuclidean2, 3117
subResultantGcd, 3117
subResultantGcdEuclidean, 3117
PSETPK, 3046
bivariate?, 3047
bivariatePolynomials, 3047
CertainlySubVariety?, 3047
crushedSet, 3047
dereduce, 3047
irreducibleFactors, 3047
lazyIrreducibleFactors, 3047
linear?, 3047
linearPolynomials, 3047
possiblyNewVariety?, 3047
probablyZeroDim?, 3047
quasiMonicPolynomials, 3047
removeIrreducibleRedundantFactors, 3047
removeRedundantFactors, 3047
removeRedundantFactorsInContents, 3047
removeRedundantFactorsInPols, 3047
removeRoughlyRedundantFactorsInContents, 3047
removeRoughlyRedundantFactorsInPols, 3047
removeSquaresIfCan, 3047
rewriteIdealWithQuasiMonicGenerators, 3047
rewriteSetByReducingWithParticularGenerators, 3047
squareFree, 3071
squareFreeFactors, 3047
ptFunc
MESH, 1552
PTFUNC2, 2966
map, 2966
POLTOPOL, 2987
ptToDmp
POLTOPOL, 2987
PTPACK, 2968
color, 2968
hue, 2968
phiCoord, 2968
rCoord, 2968
shade, 2968
thetaCoord, 2968
xCoord, 2968
yCoord, 2968
zCoord, 2968
expr2ups, 613
genups, 918
PureAlgebraicIntegration, 3137
PureAlgebraicLODE, 3146
ordfun, 2810
pushdown
pushdterm
pushFortranOutputStack
FOP, 747
pushucoef
pushuconst
mprff, 1578
IntegralBasis, 2853
PushVariables, 3148
PSEUDLIN, 3112
changeBase, 3112
companionBlocks, 3112
normalForm, 3112
pseudoDivide
PRS, 3117
PseudoLinearNormalForm, 3112
PseudoRemainderSequence, 3116
psolve
PLEQN, 2862
PSQFR, 3071
QALGSET2, 3152
INDEX

radicalSimplify, 3152
QCMPACK, 3156
   algebraicSort, 3156
   branchIfCan, 3156
   infRittWu?, 3156
   internalInfRittWu?, 3156
   internalSubPolSet?, 3156
   internalSubQuasiComponent?, 3156
   moreAlgebraic?, 3156
   prepareDecompose, 3156
   removeSuperfluousCases, 3156
   removeSuperfluousQuasiComponents, 3156
   startTable, 3156
   subCase?, 3156
   subPolSet?, 3156
   subQuasiComponent?, 3156
   subTriSet?, 3156
   supDimElseRittWu?, 3156
qfactor
   FSUPFACT, 843
QFCAT2, 3165
   map, 3165
qPot
   INBFF, 1130
qqq
   WEIER, 3732
qroot
   POLYROOT, 3042
qShiftAction
   FFFG, 756
qShiftC
   FFFG, 756
quadratic
   SOLVEFOR, 3065
quadratic?
   E04AGNT, 629
quadraticBezier
   BEZIER, 157
quadraticNorm
   GALFACTU, 848
quadTransform
   BLUPPACK, 164
quartic
   SOLVEFOR, 3065
QuasiAlgebraicSet2, 3152
QuasiComponentPackage, 3156
quasiMonicPolynomials
   PSETPK, 3047
QUATCT2, 3168
   map, 3168
QuaternioCategoryFunctions2, 3168
quickSort
   FLASORT, 729
QuotientFieldCategoryFunctions2, 3165
quotVecSpaceBasis
   LOP, 1428
Radical
   IDECOMP, 1074
   RadicalEigenPackage, 3172
   radicalEigenvalues
      REP, 3172
   radicalEigenvectors
      REP, 3172
   radicalOfLeftTraceForm
      ALGPKG, 42
   radicalRoots
      SOLVERAD, 3189
   radicalSimplify
      QALGSET2, 3152
   radicalSolve
      SOLVERAD, 3189
   RadicalSolvePackage, 3189
   radix
      RADUTIL, 3197
   RadixUtilities, 3197
   radPoly
      CHVAR, 180
   RADUTIL, 3197
      radix, 3197
   raisePolynomial
      FACUTIL, 647
   ran
      FACUTIL, 647
   randnum
      RANDSRC, 3207
   random
      FFPOLY, 695
      INBFF, 1130
   RandomDistributions, 3199
   RandomFloatDistributions, 3201
   RandomIntegerDistributions, 3204
RandomNumberSource, 3207
randomR
  GENPGCD, 904
  PFBR, 3006
  PFBRU, 3013
RANDSRC, 3207
  randnum, 3207
  reseed, 3207
  seed, 3207
  size, 3207
rangesFinite
  D01AGNT, 406
rangePascalTriangle
  GALUTIL, 873
ranges
  DROPT0, 399
rank
  LSMP, 1412
  LSMP1, 1415
  MATLIN, 1520
ratDenom
  ALGMANIP, 34
ratDsolve
  ODERAT, 3248
RATFACT, 3209
  factor, 3209
  factorSquareFree, 3209
rational
  RATRET, 3254
rational?
  RATRET, 3254
RationalFactorize, 3209
RationalFunction, 3212
RationalFunctionDefiniteIntegration, 3215
RationalFunctionFactor, 3218
RationalFunctionFactorizer, 3220
RationalFunctionIntegration, 3223
RationalFunctionLimitPackage, 3225
RationalFunctionSign, 3230
RationalFunctionSum, 3238
rationalIfCan
  RATRET, 3254
RationalIntegration, 3241
RationalInterpolation, 3244
RationalLODE, 3248
rationalPlaces
  GPAFF, 889
  PAFF, 2821
  PAFFF, 2829
rationalPoints
  GPAFF, 889
  PAFF, 2821
  PAFFF, 2829
  PRJALGPK, 2961
RationalRetractions, 3254
RationalRicDE, 3256
RationalUnivariateRepresentationPackage, 3263
ratPoly
  ALGMANIP, 34
RATRET, 3254
  rational, 3254
  rational?, 3254
  rationalIfCan, 3254
rCoord
  PPACK, 2968
RDEEF, 512
  rischDE, 512
RDEEFS, 521
  rischDEsys, 521
RDETR, 3592
  baseRDE, 3592
  monomRDE, 3592
  polyRDE, 3592
RDETRS, 3597
  baseRDEsys, 3597
  monomRDEsys, 3597
rdHack1
  RDIST, 3199
RDIST, 3199
  rdHack1, 3199
  uniform, 3199
  weighted, 3199
RDIV, 3301
  order, 3301
rdregime
  PLEQN, 2862
real
  CTRIGMNP, 248
  TRIGMNP, 3627
REAL0, 3277
  midpoint, 3277
  midpoints, 3277
INDEX

realZeros, 3277
refine, 3277
REAL0Q, 3284
realZeros, 3284
refine, 3284
refine
TRIGMNIP, 3627
realEigenvalues
NREP, 2780
realEigen vectors
NREP, 2780
realElementary
EFSTRUC, 493
RealPolynomialUtilitiesPackage, 3268
realRoots
FLOATRP, 739
REALSOLV, 3274
realSolve, 3274
solve, 3274
realSolve
REALSOLV, 3274
ZDSOLVE, 3812
RealSolvePackage, 3274
RealZeroPackage, 3277
RealZeroPackageQ, 3284
realZeros
REAL0, 3277
REAL0Q, 3284
recip
NORMPK, 2711
STTAYLOR, 3437
reciprocalPolynomial
CRFP, 233
recolor
DRAWCFUN, 3545
RECOPI, 3289
evalADE, 3289
evalRec, 3289
getEq, 3289
getOp, 3289
getShiftRec, 3289
numberOfValuesNeeded, 3289
shiftInfoRec, 3289
RectangularMatrixCategoryFunctions2, 3287
recur
MAPPKG1, 1484
RecurrenceOperator, 3289
redmat
PLEQN, 2862
REDORDER, 3306
ReduceOrder, 3306
redPo
GBINTERN, 966
redPol
GBINTERN, 966
redpps
PLEQN, 2862
reduce
ARRAY12, 2796
DEGRED, 300
DIRPRD2, 318
FLAGG2, 726
FSAGG2, 733
LIST2, 1437
MATCAT2, 1515
PRIMARR2, 3092
RMCAT2, 3287
STREAM2, 3429
VECTOR2, 3718
reducedDiscriminant
PWFFINTB, 2853
ReducedDivisor, 3301
reducedQPowers
FFPOLY, 695
reduceLineOverLine
LOP, 1428
ReduceLODE, 3304
reduceLODE
ODERED, 3304
ReduceOrder
REDORDER, 3306
reduceRow
LOP, 1428
reduceRowOnList
LOP, 1428
reduction
GENEEZ, 924
GHENSEL, 882
INMODGCD, 1113
ReductionOfOrder, 3306
refine
FRUTIL, 644
REAL0, 3277
REAL0Q, 3284

regime
PLEQN, 2862

RegularSetDecompositionPackage, 3310
RegularTriangularSetGcdPackage, 3317

removeConstantTerm
INTTOOLS, 1242

removeCoshSq
TRMANIP, 3578

removeCosSq
TRMANIP, 3578

removeIrreducibleRedundantFactors
PSETPK, 3047

removeRedundantFactors
PSETPK, 3047

removeRedundantFactorsInContents
PSETPK, 3047

removeRedundantFactorsInPols
PSETPK, 3047

removeRoughlyRedundantFactorsInContents
PSETPK, 3047

removeRoughlyRedundantFactorsInPol
PSETPK, 3047

removeRoughlyRedundantFactorsInPols
PSETPK, 3047

removeSinhSq
TRMANIP, 3578

removeSinhSq
TRMANIP, 3578

removeSquaresIfCan
PSETPK, 3047

removeSuperfluousCases
QCMPACK, 3156
SFQCMPK, 3994

removeSuperfluousQuasiComponents
QCMPACK, 3156
SFQCMPK, 3994

REP, 3172

eigenMatrix, 3172
gramschmidt, 3172
normalise, 3172
orthonormalBasis, 3172
radicalEigenvalues, 3172
radicalEigenvectors, 3172

REP1, 3330
antisymmetricTensors, 3330
createGenericMatrix, 3330
permutationRepresentation, 3330
symmetricTensors, 3330
tensorProduct, 3330

REP2, 3338
areEquivalent?, 3338
completeEchelonBasis, 3338
createRandomElement, 3338
cyclicSubmodule, 3338
isAbsolutelyIrreducible?, 3338
meatAxe, 3338
scanOneDimSubspaces, 3338
split, 3338
standardBasisOfCyclicSubmodule, 3338

REPDB, 3325
double, 3325
RepeatedDoubling, 3325
RepeatedSquaring, 3328
replaceDiffs
EXPRSOL, 593
replaceVarByOne
PFORP, 2838
replaceVarByZero
PFORP, 2838
RepresentationPackage1, 3330
RepresentationPackage2, 3338
REPSQ, 3328
expt, 3328
repSq
INBFF, 1130

reseed
RANDSRC, 3207
reset
GPAFF, 889
resetVariableOrder
UDVO, 3699
reshape
CARTEN2, 178
MPC2, 1582
RESLATC, 3355
coerce, 3355
ResolveLatticeCompletion, 3355
restorePrecision
NAGSP, 1920
INDEX

NUMODE, 2752
rk4f
NUMODE, 2752
rk4qc
NUMODE, 2752
RMCAT2, 3287
map, 3287
reduce, 3287
romberg
NUMQUAD, 2762
rombergo
NUMQUAD, 2762
rootBound
GALFACTU, 848
rootKerSimp
ALGMANIP, 34
rootNormalize
EFSTUCT, 493
rootOf
AF, 14
rootOfIrreduciblePoly
FFPOLY2, 716
rootPoly
CHVAR, 180
rootPower
ALGMANIP, 34
rootProduct
ALGMANIP, 34
rootRadius
CRFP, 233
RootsFindingPackage, 3360
rootSimp
ALGMANIP, 34
rootSplit
ALGMANIP, 34
roughBasicSet
PSETPK, 3047
rowEch
MHROWRED, 1562
rowEchelon
IMATQF, 1111
MATLIN, 1520
MHROWRED, 1562
rowEchelonLocal
MHROWRED, 1562
rowEchLocal
MHROWRED, 1562
rowEchWoZeroLines
LOP, 1428
rowEchWoZeroLinesWOVectorise
LOP, 1428
rows
MAMA, 1543
rroot
POLYROOT, 3042
RSDCMPK, 3310
algebraicDecompose, 3310
convert, 3310
decompose, 3310
internalDecompose, 3310
KrullNumber, 3310
numberOfVariables, 3310
printInfo, 3310
transcendentalDecompose, 3310
upDateBranches, 3310
RSETGCD, 3317
integralLastSubResultant, 3317
prepareSubResAlgo, 3317
startTableGcd, 3317
startTableInvSet, 3317
stopTableGcd, 3317
stopTableInvSet, 3317
toseInvertible?, 3317
toseInvertibleSet, 3317
toseLastSubResultant, 3317
toseSquareFreePart, 3317
rubiksGroup
PGE, 2945
rur
IRURPK, 1248
RURPK, 3263
RURPK, 3263
rur, 3263
s01eaf
NAGS, 2671
s13aaf
NAGS, 2671
s13acf
NAGS, 2671
s13adf
NAGS, 2671
INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX
INDEX

ARRAY12, 2796
DIRPROD2, 318
FLAGG2, 726
FSAGG2, 733
LIST2, 1437
PRIMARR2, 3092
STREAM2, 3429
VECTOR2, 3718
ScanArabic
NUMFMT, 2732
ScanFloatIgnoreSpaces
NUMFMT, 2732
ScanFloatIgnoreSpacesIfCan
NUMFMT, 2732
scanOneDimSubspaces
REP2, 3338
ScanRoman
NUMFMT, 2732
schema
PRS, 3117
schwerpunkt
CRFP, 233
SCPKG, 3466
coordinates, 3466
structuralConstants, 3466
screenResolution
GRDEF, 948
ScriptFormulaFormat1, 3368
sdf2lst
D01AGNT, 406
ESCONT, 569
ESTOOLS, 578
se2rfi
PLEQN, 2862
sec
EF, 450
STTF, 3449
STTFNC, 3460
sec2cos
TRMANIP, 3578
sech
EF, 450
sech2cosh
TRMANIP, 3578
seed
RANDSRC, 3207
SEG2, 3372
map, 3372
SEGBIND2, 3370
map, 3370
SegmentBindingFunctions2, 3370
SegmentFunctions2, 3372
selectAndPolynomials
PSETPK, 3047
selectOrPolynomials
PSETPK, 3047
selectPolynomials
PSETPK, 3047
semiDegreeSubResultantEuclidean
PRS, 3117
semiDiscriminantEuclidean
PRS, 3117
semiIndiceSubResultantEuclidean
PRS, 3117
semiLastSubResultantEuclidean
PRS, 3117
semiResultantEuclidean1
PRS, 3117
semiResultantEuclidean2
PRS, 3117
semiResultantEuclideannaif
PRS, 3117
semiResultantReducitEuclidean
PRS, 3117
semiSubResultantGcdEuclidean1
PRS, 3117
semiSubResultantGcdEuclidean2
PRS, 3117
separateDegrees
DDFACT, 328
separateFactors
DDFACT, 328
MDDFACT, 1556
sequences
PARTPERM, 2888
series
EXPR2UPS, 613
GENUPS, 918
seriesSolve
EXPRODE, 600
EXPRSOL, 593
UTSSOL, 3523
INDEX

setClipValue
  DRAWCX, 392

setErrorBound
  CRFP, 233

setFieldInfo
  INBFF, 1130

setFoundZeros
  RFP, 3360

setImagSteps
  DRAWCX, 392

setLegalFortranSourceExtensions
  FORT, 750

setOrder
  UDPO, 3696

setRealSteps
  DRAWCX, 392

setSingularPoints
  GPAFF, 889
  PAFF, 2821
  PAFFFF, 2829

setVariableOrder
  UDVO, 3699

SFQCMPK, 3394
  algebraicSort, 3394
  branchIfCan, 3394
  infRittWu?, 3394
  internalInfRittWu?, 3394
  internalSubPolSet?, 3394
  internalSubQuasiComponent?, 3394
  moreAlgebraic?, 3394
  prepareDecompose, 3394
  removeSuperfluousCases, 3394
  removeSuperfluousQuasiComponents, 3394
  startTable, 3394
  stopTable, 3394
  subCase?, 3394
  subPolSet?, 3394
  subQuasiComponent?, 3394
  subTriSet?, 3394
  supDimElseRittWu?, 3394

SFRGCD, 3411
  startTableGcd, 3411
  stopTableGcd, 3411
  stopTableInvSet, 3411
  stoseIntegralLastSubResultant, 3411
  stoseInternalLastSubResultant, 3411
  stoseInvertible?, 3411
  stoseInvertible?reg, 3411
  stoseInvertible?sqfreg, 3411
  stoseInvertibleSet, 3411
  stoseInvertibleSetreg, 3411
  stoseInvertibleSetsqfreg, 3411
  stoseLastSubResultant, 3411
  stosePrepareSubResAlgo, 3411
  stoseSquareFreePart, 3411

SFunction
  CYCLES, 281
  SGCF, 3498
    coleman, 3498
    inverseColeman, 3498
    listYoungTableaus, 3498
    makeYoungTableau, 3498
    nextColeman, 3498
    nextLatticePermutation, 3498
    nextPartition, 3498
    numberOfImproperPartitions, 3498
    subSet, 3498
    unrankImproperPartitions0, 3498
    unrankImproperPartitions1, 3498

shade
  PTPACK, 2968
  shanksDiscLogAlgorithm
  DLP, 321
  shellSort
  FLASORT, 729

ShiftAction
  FFFG, 756

shiftHP
  GUESS, 1012
  GUESSAN, 1048
  GUESSF, 1050
  GUESSINT, 1053
  GUESSP, 1055
  GUESSUP, 1057

shiftInfoRec
INDEX

RECOP, 3289
shiftRoots
   GALPOLYU, 870
showFortranOutputStack
   FOP, 747
SHP, 3470
   countRealRoots, 3470
countRealRootsMultiple, 3470
SturmHabicht, 3470
SturmHabichtCoefficients, 3470
SturmHabichtMultiple, 3470
SturmHabichtSequence, 3470
subresultantSequence, 3470
shuffle
   PARTPERM, 2888
shufflein
   PARTPERM, 2888
Si
   LF, 1432
sign
   SIGNEF, 487
   SIGNRF, 3230
   TOOLSIGN, 3531
signAround
   INPSIGN, 1149
SIGNEF, 487
   sign, 487
SIGNRF, 3230
   sign, 3230
SIMPAN, 3376
   simplify, 3376
SimpleAlgebraicExtensionAlgFactor, 3374
simpleBounds?
   E04AGNT, 629
simplify
   SIMPAN, 3376
   TRMANIP, 3578
SimplifyAlgebraicNumberConvertPackage, 3376
simplifyExp
   TRMANIP, 3578
simplifyLog
   TRMANIP, 3578
simpson
   NUMQUAD, 2762
simpsono
   NUMQUAD, 2762
sin
   EF, 450
   STTF, 3449
   STTFNC, 3460
sin2csc
   TRMANIP, 3578
sincos
   STTF, 3449
singleFactorBound
   GALFACTU, 848
singRicDE
   ODEPRRIC, 3103
   ODERTRIC, 3256
singularitiesOf
   D01AGNT, 406
   ESCONT, 569
singularPoints
   GPAFF, 889
   PAFF, 2821
   PAFFFF, 2829
   PRJALGPK, 2961
singularPointsWithRestriction
   PRJALGPK, 2961
sinh
   EF, 450
   sinh2csc
   TRMANIP, 3578
sinhcosh
   STTF, 3449
size
   RANDSRC, 3207
sizeMultiplication
   FFF, 680
sizePascalTriangle
   GALUTIL, 873
skewSFunction
   CYCLES, 281
   TABLBUMP, 3512
slope
   NPOLYGON, 2692
SMITH, 3378
   completeHermite, 3378
   completeSmith, 3378
diophantineSystem, 3378
   hermite, 3378
INDEX

smith, 3378
smith
SMITH, 3378
SmithNormalForm, 3378
sn
ELFUTS, 524
sncndn
ELFUTS, 524
socf2socdf
ESTOOLS, 578
solve
FLOATRP, 739
LODEEF, 474
LSMP, 1412
LSMP1, 1415
NLINSOL, 2708
NODE1, 2703
ODEEF, 481
ODEPACK, 77
ODESYS, 3485
PDEPACK, 86
REALSOLV, 3274
SOLVEFOR, 3065
SOLVETRA, 3608
SYSSOLP, 3492
solveid
GENEEZ, 924
solveInField
NLINSOL, 2708
ODESYS, 3485
solveLinear
LINDEP, 1396
solveLinearlyOverQ
ZLINDEP, 1182
solveLinearPolynomialEquation
CINTSLE, 226
FFSLE, 720
INTSLE, 1221
solveLinearPolynomialEquationByFractions
LPEFRAC, 1406
solveLinearPolynomialEquationByRecursion
PFBR, 3006
PFBRU, 3013
SOLVERAD, 3189
contractSolve, 3189
radicalRoots, 3189
radicalSolve, 3189
solveRetract
RETSOL, 3357
SOLVEVESR, 3620
decomposeFunc, 3620
unvectorise, 3620
SOLVEVTRA, 3608
solve, 3608
sortConstraints
E04AGNT, 629
SortedCache, 3384
SortPackage, 3387
SORTPAK, 3387
bubbleSort, 3387
insertionSort, 3387
space
DROPT0, 399
SparseUnivariatePolynomialFunctions2, 3389
sparsityIF
D02AGNT, 420
SpecialOutputPackage, 3391
specialTrigs
EF, 450
SPECOUT, 3391
outputAsFortran, 3391
outputAsScript, 3391
outputAsTex, 3391
spherical
COORDSYS, 255
split
ALGFACKT, 52
IR2F, 1236
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRRF2F</td>
<td>1233</td>
</tr>
<tr>
<td>MONOTOOL</td>
<td>1570</td>
</tr>
<tr>
<td>REP2</td>
<td>3338</td>
</tr>
<tr>
<td>splitConstant</td>
<td></td>
</tr>
<tr>
<td>INTPM</td>
<td>2908</td>
</tr>
<tr>
<td>splitDenominator</td>
<td></td>
</tr>
<tr>
<td>COMBF</td>
<td>211</td>
</tr>
<tr>
<td>ICDEN</td>
<td>1102</td>
</tr>
<tr>
<td>MCDEN</td>
<td>1517</td>
</tr>
<tr>
<td>ODEPRIM</td>
<td>3098</td>
</tr>
<tr>
<td>UPCDEN</td>
<td>3664</td>
</tr>
<tr>
<td>splitLinear</td>
<td></td>
</tr>
<tr>
<td>E04AGNT</td>
<td>629</td>
</tr>
<tr>
<td>splitSquarefree</td>
<td></td>
</tr>
<tr>
<td>MONOTOOL</td>
<td>1570</td>
</tr>
<tr>
<td>sPol</td>
<td></td>
</tr>
<tr>
<td>GBINTERN</td>
<td>966</td>
</tr>
<tr>
<td>sqfree</td>
<td></td>
</tr>
<tr>
<td>PLEQN</td>
<td>2862</td>
</tr>
<tr>
<td>squareFree</td>
<td></td>
</tr>
<tr>
<td>INTFACT</td>
<td>1172</td>
</tr>
<tr>
<td>MULTSQFR</td>
<td>1615</td>
</tr>
<tr>
<td>PSQFR</td>
<td>3071</td>
</tr>
<tr>
<td>SUPFRACF</td>
<td>3483</td>
</tr>
<tr>
<td>UPSQFREE</td>
<td>3678</td>
</tr>
<tr>
<td>ZDSOLVE</td>
<td>3812</td>
</tr>
<tr>
<td>squareFreeFactors</td>
<td></td>
</tr>
<tr>
<td>PSETPK</td>
<td>3047</td>
</tr>
<tr>
<td>squareFreeLexTriangular</td>
<td></td>
</tr>
<tr>
<td>LEXTRIPK</td>
<td>1390</td>
</tr>
<tr>
<td>squareFreePart</td>
<td></td>
</tr>
<tr>
<td>UPSQFREE</td>
<td>3678</td>
</tr>
<tr>
<td>squareFreePrim</td>
<td></td>
</tr>
<tr>
<td>MULTSQFR</td>
<td>1615</td>
</tr>
<tr>
<td>SquareFreeQuasiComponentPackage</td>
<td>3394</td>
</tr>
<tr>
<td>SquareFreeRegularSetDecompositionPackage</td>
<td>3404</td>
</tr>
<tr>
<td>SquareFreeRegularTriangularSetGcdPackage</td>
<td>3411</td>
</tr>
<tr>
<td>SRDCMPK</td>
<td>3404</td>
</tr>
<tr>
<td>algebraicDecompose</td>
<td>3404</td>
</tr>
<tr>
<td>convert</td>
<td>3404</td>
</tr>
<tr>
<td>decompose</td>
<td>3404</td>
</tr>
<tr>
<td>internalDecompose</td>
<td>3404</td>
</tr>
<tr>
<td>KrullNumber</td>
<td>3404</td>
</tr>
<tr>
<td>numberOfVariables</td>
<td>3404</td>
</tr>
<tr>
<td>printInfo</td>
<td>3404</td>
</tr>
<tr>
<td>transcendentalDecompose</td>
<td>3404</td>
</tr>
<tr>
<td>upDateBranches</td>
<td>3404</td>
</tr>
<tr>
<td>standardBasisOfCyclicSubmodule</td>
<td></td>
</tr>
<tr>
<td>REP2</td>
<td>3338</td>
</tr>
<tr>
<td>startPolynomial</td>
<td></td>
</tr>
<tr>
<td>CRFP</td>
<td>233</td>
</tr>
<tr>
<td>stepBlowUp</td>
<td></td>
</tr>
<tr>
<td>BLUPPACK</td>
<td>164</td>
</tr>
<tr>
<td>stFunc1</td>
<td></td>
</tr>
<tr>
<td>UTSODE</td>
<td>3690</td>
</tr>
<tr>
<td>stFunc2</td>
<td></td>
</tr>
<tr>
<td>UTSODE</td>
<td>3690</td>
</tr>
<tr>
<td>stFuncN</td>
<td></td>
</tr>
<tr>
<td>UTSODE</td>
<td>3690</td>
</tr>
<tr>
<td>stiffnessAndStabilityFactor</td>
<td></td>
</tr>
<tr>
<td>D02AGNT</td>
<td>420</td>
</tr>
<tr>
<td>stiffnessAndStabilityOfODEIF</td>
<td></td>
</tr>
<tr>
<td>D02AGNT</td>
<td>420</td>
</tr>
<tr>
<td>STINPROD</td>
<td>3434</td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>generalInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>stirling1</td>
<td></td>
</tr>
<tr>
<td>COMBINAT</td>
<td>1167</td>
</tr>
<tr>
<td>stirling2</td>
<td></td>
</tr>
<tr>
<td>COMBINAT</td>
<td>1167</td>
</tr>
<tr>
<td>STINPROD</td>
<td>3434</td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>generalInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>stirling1</td>
<td></td>
</tr>
<tr>
<td>COMBINAT</td>
<td>1167</td>
</tr>
<tr>
<td>STINPROD</td>
<td>3434</td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>generalInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
<td>3434</td>
</tr>
<tr>
<td>GAFACT</td>
<td>853</td>
</tr>
<tr>
<td>StorageEfficientMatrixOperations</td>
<td>3422</td>
</tr>
<tr>
<td>stoseIntegralLastSubResultant</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
<tr>
<td>stoseInternalLastSubResultant</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
<tr>
<td>stoseInvertible?</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
<tr>
<td>stoseInvertible?sqfreg</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
<tr>
<td>stoseInvertibleSet</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
<tr>
<td>stoseInvertibleSetreg</td>
<td></td>
</tr>
<tr>
<td>SFRGCD</td>
<td>3411</td>
</tr>
</tbody>
</table>
INDEX

SFRGCD, 3411
stoseInvertibleSetsqreg
SFRGCD, 3411
stoseLastSubResultant
SFRGCD, 3411
stosePrepareSubResAlgo
SFRGCD, 3411
stoseSquareFreePart
SFRGCD, 3411
STREAM1, 3427
concat, 3427
STREAM2, 3429
map, 3429
reduce, 3429
scan, 3429
STREAM3, 3432
map, 3432
StreamFunctions1, 3427
StreamFunctions2, 3429
StreamFunctions3, 3432
StreamInfiniteProduct, 3434
StreamTaylorSeriesOperations, 3437
STREAMTensor, 3447
StreamTensor, 3447
StreamTranscendentalFunctions, 3449
StreamTranscendentalFunctionsNonCommutative, 3460
stripCommentsAndBlanks
TEMUTL, 3527
structuralConstants
SCPKG, 3466
StructuralConstantsPackage, 3466
sts2stst
WEIER, 3732
STTAYLOR, 3437
-?, 3437
**?, 3437
?+?, 3437
?-, 3437
/?, 3437
adddiag, 3437
coece, 3437
compose, 3437
deriv, 3437
eval, 3437
evenlambert, 3437
exquo, 3437
gderiv, 3437
generalLambert, 3437
int, 3437
integers, 3437
integrate, 3437
invmultisect, 3437
lagrange, 3437
lambert, 3437
lazyGintegrate, 3437
lazyIntegrate, 3437
mapdiv, 3437
mapmult, 3437
monom, 3437
multisect, 3437
nlde, 3437
oddintegers, 3437
oddlambert, 3437
power, 3437
powern, 3437
recip, 3437
revert, 3437
STTF, 3449
?**?, 3449
acos, 3449
acot, 3449
acsc, 3449
asec, 3449
asin, 3449
atan, 3449
cos, 3449
cot, 3449
csc, 3449
csc, 3449
d, 3449
exp, 3449
sec, 3449
sin, 3449
sincos, 3449
sinhcosh, 3449
tan, 3449
atan, 3449
STTFNC, 3460
?**?, 3460
acos, 3460
acot, 3460
acsc, 3460
asec, 3460
asin, 3460
atan, 3460
cos, 3460
cot, 3460
csc, 3460
exp, 3460
sec, 3460
sin, 3460
tan, 3460
SturmHabicht
SHP, 3470
SturmHabichtCoefficients
SHP, 3470
SturmHabichtMultiple
SHP, 3470
SturmHabichtPackage, 3470
SturmHabichtSequence
SHP, 3470
sturmSequence
POLUTIL, 3268
sturmVariationsOf
POLUTIL, 3268
style
DROPTO, 399
subCase?
QCMPACK, 3156
SFQCMKP, 3394
subMatrix
MAMA, 1543
subPolSet?
QCMPACK, 3156
SFQCMKP, 3394
subQuasiComponent?
QCMPACK, 3156
SFQCMKP, 3394
SUBRESP, 3479
primitivePart, 3479
subresultantVector, 3479
subResultantGcd
PRS, 3117
subResultantGcdEuclidean
PRS, 3117
SubResultantPackage, 3479
subresultantSequence
SHP, 3470
subresultantVector
SUBRESP, 3479
subs1stVar
PFORP, 2838
subs2ndVar
PFORP, 2838
subscriptedVariables
D03AGNT, 427
subSet
SGCF, 3498
subsInVar
PFORP, 2838
subTriSet?
QCMPACK, 3156
SFQCMKP, 3394
suchThat
PATTERN1, 2892
PMPRED, 113
PMPREDFS, 793
sum
ISUMP, 1151
SUMFS, 811
SUMRF, 3238
SUMFS, 811
sum, 811
summation
COMBF, 198
sumOfDivisors
INTHEORY, 1197
sumOfKthPowerDivisors
INTHEORY, 1197
sumOfSquares
E04AGNT, 629
SUMRF, 3238
sum, 3238
sumSquares
GAUSSFAC, 877
SUP2, 3389
map, 3389
supDimElseRittWu?
QCMPACK, 3156
SFQCMKP, 3394
SUPFRACF, 3483
factor, 3483
squareFree, 3483
SupFractionFactorizer, 3483
swap
COMMUPC, 219
sylvesterMatrix
BEZOUT, 159
sylvesterSequence
POLUTIL, 3268
SYMFUNC, 3509
symFunc, 3509
symFunc
SYMFUNC, 3509
SymmetricFunctions, 3509
symmetricGroup
PGE, 2945
SymmetricGroupCombinatoricFunctions, 3498
symmetricPower
LODOOPS, 1403
symmetricProduct
LODOOPS, 1403
symmetricTensors
REPL, 3330
SYSSOLP, 3492
solve, 3492
triangularSystems, 3492
systemCommand
MSYSCMD, 1573
SystemODESolver, 3485
systemSizeIF
D02AGNT, 420
SystemSolvePackage, 3492
t
RFDIST, 3201
tab
TABLBUMP, 3512
tab1
TABLBUMP, 3512
TABLBUMP, 3512
bat, 3512
bat1, 3512
bumprow, 3512
bumptab, 3512
bumptab1, 3512
inverse, 3512
lex, 3512
maxrow, 3512
mr, 3512
slex, 3512
tab, 3512
tab1, 3512
untab, 3512
TableauxBumpers, 3512
tablePow
GENEEZ, 924
TabulatedComputationPackage, 3516
tan
EF, 450
STTF, 3449
STTTFNC, 3460
tan2cot
TRMANIP, 3578
tan2trig
TRMANIP, 3578
tanAn
TANEXP, 3520
TANEXP, 3520
tanAn, 3520
tanNa, 3520
tanSum, 3520
TangentExpansions, 3520
tanh
EF, 450
tanh2coth
TRMANIP, 3578
tanh2trigh
TRMANIP, 3578
tanintegrate
INTTR, 3567
tanNa
TANEXP, 3520
tanQ
EFSTRUC, 493
tanSum
TANEXP, 3520
taylor
EXPR2UPS, 613
GENUPS, 918
TaylorSolve, 3523
TBCMPPK, 3516
clearTable, 3516
extractIfCan, 3516
initTable, 3516
insert, 3516
makingStats?, 3516
printInfo, 3516
printingInfo?, 3516
printStats, 3516
startStats, 3516
usingTable?, 3516
TemplateUtilities, 3527
TEMUTL, 3527
interpretString, 3527
stripCommentsAndBlanks, 3527
tensorMap
    STNSR, 3448
tensorProduct
    REP1, 3330
testDim
    GROEBSOL, 1006
testModulus
    GENEEX, 924
TEXI, 3529
    coerce, 3529
TexFormat1, 3529
theCurve
    GPAFF, 889
    PAFF, 2821
    PAFFFF, 2829
thetaCoord
    PT PACK, 2968
times
    OREPCTO, 3684
title
    DROPTO, 399
tomodpa
    POLYVEC, 3704
ToolsForSign, 3531
TOOLSIGN, 3531
direction, 3531
    nonQsign, 3531
    sign, 3531
topFortranOutputStack
    FOP, 747
TopLevelDrawFunctions, 3533
TopLevelDrawFunctionsForAlgebraicCurves, 3533
TopLevelDrawFunctionsForCompiledFunctions, 3545
TopLevelDrawFunctionsForPoints, 3559
TopLevelThreeSpace, 3562
TOPSP, 3562
createThreeSpace, 3562
toroidal
    COORDSYS, 255
torsion?
    PFO, 2971
    PFOQ, 2978
torsionIfCan
    PFO, 2971
    PFOQ, 2978
toScale
    DROPTO, 399
toseInvertible?
    RSETGCD, 3317
toseInvertibleSet
    RSETGCD, 3317
toseLastSubResultant
    RSETGCD, 3317
toseSquareFreePart
    RSETGCD, 3317	otalDegree
    PFORP, 2838
totalFrac
    MPRFF, 1578
totalGroebner
    PGROEB, 2990	totalLex
    ORDFUNS, 2810
totolex
    LGROBP, 1420
trace
    INBF F, 1130
    trace2PowMod
        DDFAC T, 328
    tracePowMod
        DDFACT, 328
transcendentalDecompose
    RSDCMPK, 3310
    SRDCMPK, 3404
TranscendentalHermiteIntegration, 3564
TranscendentalIntegration, 3567
TranscendentalManipulations, 3577
TranscendentalRischDE, 3592
TranscendentalRischDESystem, 3597	transform
    LGROBP, 1420
translate
    PFORP, 2838
translateToOrigin
INDEX

PAFFF, 2829
PLPKCRV, 2984
TransSolvePackage, 3608
TransSolvePackageService, 3620
trapezoidal
  NUMQUAD, 2762
trapezoidal0
  NUMQUAD, 2762
triangSolve
  ZDSOLVE, 3812
TriangularMatrixOperations, 3624
triangularSystems
  SYSSOLP, 3492
triangulate
  ODESYS, 3485
TRIGMNIP, 3627
  complexElementary, 3627
  complexForm, 3627
  complexNormalize, 3627
  imag, 3627
  real, 3627
  real?, 3627
  trigs, 3627
TrigonometricManipulations, 3627
  trigs
    CTRIGMNP, 248
    TRIGMNIP, 3627
  trigs2explogs
    ITRIGMNIP, 1154
TRIMAT, 3624
  LowTriBddDenomInv, 3624
  UpTriBddDenomInv, 3624
TRMANIP, 3577
  cos2sec, 3578
  cosh2sech, 3578
  cot2tan, 3578
  cot2trig, 3578
  coth2tanh, 3578
  coth2trigh, 3578
  csch2sinh, 3578
  csch2sinh, 3578
  expand, 3578
  expandLog, 3578
  expandPower, 3578
  expandTrigProducts, 3578
  htrigs, 3578
  removeCoshSq, 3578
  removeCosSq, 3578
  removeSinhSq, 3578
  removeSinSq, 3578
  sec2cos, 3578
  sech2cosh, 3578
  simplify, 3578
  simplifyExp, 3578
  simplifyLog, 3578
  sin2csc, 3578
  sinh2csch, 3578
  tan2cot, 3578
  tan2trig, 3578
  tanh2coth, 3578
  tanh2trigh, 3578
  truncatedmuladd
    POLYVEC, 3704
trunancedmultiplication
    POLYVEC, 3704
tryFunctionalDecomposition
  GALFACT, 853
tryFunctionalDecomposition?
  GALFACT, 853
tube
  NUMTUBE, 2782
tubePlot
    EXPRTUBE, 621
    TubePlotTools, 3631
tubePoints
    DROPT0, 399
tubePointsDefault
    VIEWDEF, 3720
tubeRadius
    DROPT0, 399
tubeRadiusDefault
    VIEWDEF, 3720
TUBETOOL, 3631
  *.*, 3631
  +., 3631
  -.*, 3631
cosSinInfo, 3631
cross, 3631
dot, 3631
loopPoints, 3631
point, 3631
unitVector, 3631
twist
  MAPPKG3, 1505
TwoDimensionalPlotClipping, 3635
TWOFACT, 3642
  generalSqFr, 3642
generalTwoFactor, 3642
twoFactor, 3642
twoFactor
  TWOFACT, 3642
TwoFactorize, 3642
U32VectorPolynomialOperations, 3704
UDPO, 3696
  getOrder, 3696
  largest, 3696
  less?, 3696
  more?, 3696
  setOrder, 3696
  userOrdered?, 3696
UDVO, 3699
  getVariableOrder, 3699
  resetVariableOrder, 3699
  setVariableOrder, 3699
UFPS1, 3658
  hadamard, 3658
ULS2, 3660
  map, 3660
unaryFunction
  MKUCFUNC, 1468
uncouplingMatrices
  ASSOCEQ, 110
UNIFACT, 3650
  factor, 3650
  factorSquareFree, 3650
  henselFact, 3650
uniform
  RDIST, 3199
  RFDIST, 3201
  RIDIST, 3204
uniform01
  RFDIST, 3201
union
  INTTOOLS, 1242
UNISEG2, 3694
  map, 3694
units
  DROPT0, 399
  unitsColorDefault
    VIEWDEF, 3720
  unitVector
    TUBETOOL, 3631
univariate
  INTG0, 932
  PFORP, 2838
  POLY2UP, 3075
  POLYCATQ, 2997
  RF, 3212
univariate?
  PSETPK, 3047
UnivariateFactorize, 3650
UnivariateFormalPowerSeriesFunctions, 3658
UnivariateLaurentSeriesFunctions2, 3660
UnivariatePolynomialCategoryFunctions2, 3662
UnivariatePolynomialCommonDenominator, 3664
UnivariatePolynomialDecompositionPackage, 3666
UnivariatePolynomialDivisionPackage, 3670
UnivariatePolynomialFunctions2, 3672
UnivariatePolynomialMultiplicationPackage, 3674
UnivariatePolynomials
  PSETPK, 3047
univariatePolynomialsGcds
  PSETPK, 3047
UnivariatePolynomialSquareFree, 3678
UnivariatePuiseuxSeriesFunctions2, 3681
UnivariateSkewPolynomialCategoryOps, 3684
univariateSolve
  ZDSOLVE, 3812
UnivariateTaylorSeriesFunctions2, 3688
UnivariateTaylorSeriesODESolver, 3690
univcase
  MULTSQFR, 1615
UniversalSegmentFunctions2, 3694
unprotectedRemoveRedundantFactors
  PSETPK, 3047
unrankImproperPartitions0
  SGCF, 3498
unrankImproperPartitions1
  SGCF, 3498
untab
  TABLBUMP, 3512
unvectorise
INTTOOLS, 1242
varList
   D03AGNT, 427
   E04AGNT, 629
vselect
   INTTOOLS, 1242
VECTOR2, 3718
   map, 3718
   reduce, 3718
   scan, 3718
vectoraddmul
   POLYVEC, 3704
vectorcombination
   POLYVEC, 3704
VectorFunctions2, 3718
vedf2vef
   ESTOOLS, 578
vertConcat
   MAMA, 1543
vertSplit
   MAMA, 1543
VIEW, 3726
   coerce, 3726
   drawCurves, 3726
   graphCurves, 3726
VIEWDEF, 3720
   axesColorDefault, 3720
   lineColorDefault, 3720
   pointColorDefault, 3720
   pointSizeDefault, 3720
   tubePointsDefault, 3720
   tubeRadiusDefault, 3720
   unitsColorDefault, 3720
   var1StepsDefault, 3720
   var2StepsDefault, 3720
   viewDefaults, 3720
   viewPosDefault, 3720
   viewSizeDefault, 3720
   viewWriteAvailable, 3720
   viewWriteDefault, 3720
viewPosDefault
   VIEWDEF, 3720
viewSizeDefault
   VIEWDEF, 3720
viewWriteAvailable
   VIEWDEF, 3720
viewWriteDefault
   VIEWDEF, 3720
virtualDegree
   GBINTERN, 966
weakBiRank
   ALGPKG, 42
WEIER, 3732
   cfirstrst, 3732
   clikelUniv, 3732
   cresst, 3732
   qqq, 3732
   stst2stst, 3732
   weierstrass, 3732
weierstrass
   WEIER, 3732
   WeierstrassPreparation, 3732
weighted
   RDIST, 3199
WFFINTBS, 3736
   integralBasis, 3736
   localIntegralBasis, 3736
WildFunctionFieldIntegralBasis, 3736
wreath
   CYCLES, 281
writeObj
   EXP3D, 626
wronskianMatrix
   ODETOOLS, 2793
wrregime
   PLEQN, 2862
xCoord
   PTPACK, 2968
XExponentialPackage, 3744
XEXPPKG, 3744
   exp, 3744
   Hausdorff, 3744
   log, 3744
xii
INDEX

INBFF, 1130

Y
Y STREAM, 2859
y Coord
PTPACK, 2968
young Group
PGE, 2945
Y STREAM, 2859
Y, 2859
Yun
FFSQFR, 722

z Coord
PTPACK, 2968
ZDSOLVE, 3812
convert, 3812
positive Solve, 3812
real Solve, 3812
square Free, 3812
triang Solve, 3812
univariate Solve, 3812
zero Dimensional?
FGLM IC PK, 658
LEXTRIPK, 1390
Zero Dimensional Solve Package, 3812
zero Dim Primary?
IDECOMP, 1074
zero Dim Prime?
IDECOMP, 1074
zero Matrix
FCPAK1, 743
zero Set Split
LAZM3PK, 1313
LEXTRIPK, 1390
zeros Of
ESCONT, 569
zero Square Matrix
FCPAK1, 743
zero Vector
FCPAK1, 743
Zeta Function
GPAFF, 889
PAFF, 2821
PAFFFF, 2829
ZLINDEP, 1182
lin ear Dependence Over Z, 1182
linearly Dependent Over Z?, 1182
solve Linearly Over Q, 1182